A Service-Based Component Model: Formalism, Analysis and Mechanization

Pascal André, Gilles Ardourel, Christian Attiogbé, Henri Habrias, Cédric Stoquer

A Service-Based Component Model: Formalism, Analysis and Mechanization

P. André, G. Ardourel, C. Attiogbé, H. Habrias, C. Stoquer

Laboratoire d'Informatique de Nantes-Atlantique
2, rue de la Houssinière - B.P. 92208 - 44322 NANTES CEDEX 3

— Components, Services, Behavioural Interface Description, Interaction Checking —

RESEARCH REPORT

N° 05.08

December 2005

P. André, G. Ardourel, C. Attiogbé, H. Habrias, C. Stoquer

A Service-Based Component Model: Formalism, Analysis and Mechanization

70 p.

A Service-Based Component Model: Formalism, Analysis and Mechanization

P. André, G. Ardourel, C. Attiogbé, H. Habrias, C. Stoquer

Prenom.Nom@univ-nantes.fr

Abstract

Component-Based Software Engineering (CBSE) is one of the approaches to master the development of large scale software. In this setting, the verification concern is still a challenge. The objective of our work is to provide the designer of components-based systems with the methods to assist his/her use of the components. In particular, the current work adresses the composability of components and their services.
A component model is presented, based on services. An associated simple but expressive formalism is introduced; it describes the services as extended LTS and their structuring as components. The composition of components is mainly based on service composition and encapsulation.
The composability of component is defined from the composability of services. To ensure the correctness of component composition, we check that an assembly is possible via the checking of the composabiblity of the linked services, and their behavioral compatibility. In order to mechanize our approach, the services and the components are translated into the MEC and LOTOS formalism. Finally the MEC and LOTOS CADP toolbox is used to perform experiments.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; D.2.11 [**Software Engineering**]: Software Architectures—*Languages*

General Terms: Components, Services, Behavioural Interface Description, Interaction Checking

Additional Key Words and Phrases: Components, Services, Behavioural Interface Description, Interaction Checking

Contents

This report is a collection of idea, studies and development that mark a step of an ongoing work. It is considered as a draft version; therefore it is submitted to important modifications on each section.

1 Introduction

The rigorous development of (correct) large systems with methods that scale up and are reusable in various projects is still a challenging research topic. From this point of view Component-Based Software Engineering (CBSE) motivates a number of works [32, 23, 11, 19]. Component-Based Software Engineering promotes the (re)use of components coming from third party developers to build new large systems. But this raises some challenging problems and many of them are still open issues. The success of the large scale development of component-based systems depends the availability of:

- reliable components library,
- tools to search for components (in libraries),
- expressive languages of composition of the components and especially tools for checking the good use of components.

It is important to detect the defects which could lead to a faulty behaviour of the developed system early in the development. A bad interaction between a called service and the appealing one (from a component) may lead to a blocking of the whole system. To ensure a good level of correctness of the components and their assemblies, the formal verification of the service descriptions with respect to the desired properties of the component is necessary; tests of conformity of the interaction between the components must also be carried out at the time of their assembly. One can thus guarantee the good functioning of an assembly. Consequently, the specifications of components and their service behaviours should be abstract and formal. The use of an abstract formal model also makes it possible to hide the implementation details of the components in order to have general reasoning techniques which are adaptable to various implementation environments.

The goal of our work is to provide the designer of component-based systems with the methods to assist his/her use of the components. But we focus more specifically on the last parameter given above: search of an expressive language accompanied with an experimental toolbox for component study and development.

The general *motivation* for this work lies on the need of a sound basis for developping correct components and for studying component composition and for implementing the related tools. Indeed, many works focus on the behavioural compatibility and deal with simplified abstract component models [11, 15, 10]. On the other hand there are mechanized approaches such as the Tracta approach [17] or the SOFA approach [26, 27] but their component models have some limitations. The use of one behaviour protocol is an example of such limitation.

For this purpose, our approach is based on a model (named Kmelia) supported by a simple formalism for modeling and composing components. A component is viewed and used through the *services* which constitute its interface; each service has a behavioural interface. Components are assembled and composed through their services. The use of services is central to the verification of composability when assembling components. We define composability of components by considering the links between their services and the behavioural compatibility of these services. Therefore, we have the basis for the study of component assemblies and compositions. We distinguish four levels of conformity test:

- service signatures as in IDL,
- enhanced service signature (when considering that sub-services can be participant of a service),
- contracts (pre/post conditions)
- and behaviour (the interactions -waiting for data, synchronization- between the caller service and the called service are correct).

When the matching of the signature of a service at the time of its call seems a pre-requisite easily verifiable, the *a priori* conformity test of the interaction between a service and its appealing, during the running of the service, is not an easy task. Indeed, the interaction can be either very simple and may resumes to a "call-answer" or it can be more complex, when the running of the service requires the collaboration of the appealing one. In addition to the signatures, the *description of the behaviours* of the services is necessary to the appealing ones. That means the requirements (waiting for data, synchronization, etc) of one are satisfied by the other and vice versa. In this work, we explore the means of rigorously carrying out these checks of conformity of interactions by studying a formal model of components. Practically a behavioural conformity check will make it possible to detect the incompatibilities of behaviour between components (or services) which interact. For example, a service awaits a value whereas the other cannot provide some in this state of their joint evolution.

In summary, the contribution of the current work is first, a model for describing component, services, assemblies and compositions and second a toolbox for verifying the behavioural compatibility of component assemblies. This work is supported by a prototype, called `COSTO` (COmponent Study TOolbox). The prototype under development is currently made of a Kmelia analyser and two translation tools `kmelia2mec` and `kmelia2lotos`.

This research report shows the current state of the work. It is structured as follows. Section 2 presents the Kmelia model through the description of services, components and composition. This is illustrated with an example of a bank Automatic Teller Machine (ATM) system in Section 3. The Section 4 presents the composability of services and the composability of components. The latter are used to analyse component assembly and component composition. Behavioural compatibility between component services is also treated there. In the Section 6 and the Section 5, we present the mechanization approach undertaken to support the Kmelia model. Experiments are done with MEC4 and LOTOS CADP. The Section 8 compares our approach with related works and the Section 9 concludes the report.

2 A Formalism for Specifying Components, Interfaces and Assemblies

In the Kmelia model, a component is characterised by: its name (the component identifier); its state (variables and an invariant predicate on them); its interface made of *services* provided and required by the component and the description of the services which constitute the component behaviour(s). Component are either elementary (with no references to other components) or defined by assembling other components. Assemblies and composition are described in Section 2.3. A preliminary version of the model is presented in [4].

As usually [2, 23] the *interface* specifies the component interactions with its environment. We assume that the service executions are concurrent processes with shared (component) state. But unlike most of the existing approaches [27, 31, 25] where the only unit of interaction is a *message*, we also consider *services* as units of interaction. Therefore whithin our model a component interface is made of *provided services* and *required services*. A provided service offers a functionality, while a required service is the expression of the need of a functionality. This need is satisfied when the component is combined with other components (in an *assembly*), one of them supplying the corresponding provided service. As the service is central in our approach, we specify it before the component in this section.

2.1 Service Specification

A *service* is defined by an *interface* and a *behaviour* which specifies the dynamic evolution. A service s of a component C is specified with $\langle I_s, \mathcal{B}_s \rangle$ where I_s is the service interface, and $\mathcal{B}_s$ is the *extended labelled transition system* (eLTS) which specifies the service behaviour. A required service does not need the same level of detail as a provided service. Therefore the former does not have a behaviour specification and may not have a pre/post condition.

2.1.1 Service Interface

The interface I_s of a service s is defined by a 5-tuple $\langle \sigma,\ P,\ Q,\ V_s, S_s \rangle$ where:

- σ is the service signature,

- P is a precondition,
- Q is a postcondition,
- V_s is a set of local declarations and
- the *service dependency* S_s is a 4-tuple $S_s = \langle sub_s,\ cal_s,\ req_s,\ int_s \rangle$ of disjoint sets.

Service Dependencies
sub_s (resp. cal_s, req_s, int_s) is a set of the provided services names (resp. the services required from the caller, the services required from any component, the internal provided services) in the scope s. Using a required service r in cal_s of a service p (as opposed to a component interface) means that r should be provided by the component which calls p. By having a provided service p in the sub_s of a service r but not in the component interface, we express that p is accessible only during an interaction with r.

2.1.2 Service Behaviour

The behaviour $\mathcal{B}_s$ of a service s is defined by a 6-tuple $\langle S, L, \delta, \Phi, S_0, S_F \rangle$ with

- S the set of the states of s;
- L is the set of transition labels ;
- $\delta \in S \times L \rightarrow S$ is the transition relation ;
- $S_0 \in S$ is the initial state ;
- $S_F \subseteq S$ is the finite set of final states ;
- $\Phi : S \rightarrow sub_s$ is a state anotation function.

Branching State
An eLTS is obtained when we allow *branching states* among the states of an LTS (using the Φ function). A branching state is the one annotated with sub-service names, which are (sub-)services of the component C that may be called when the evolution reaches this state (but the control returns to this state when the launched sub-service is terminated). This provides description flexibility. In the current version of Kmelia, only one type of branching states is allowed, the optional (sub-)services call. But other forms are possible, like conditional branching states, mandatory branching states. The latter is in fact associated to transitions, as branching transitions. Formally, the unfolding of (the branching states and transitions of) an eLTS results in an LTS.

Transitions
The elements of δ have the abstract shape $(ss, label, ts)$ or the concrete Kmelia syntax `ss--label-->ts`. The labels are (possibly guarded) combinations[1] of actions: `[guard] action*`. We assume the following (restricted) usage: for any states, the outgoing transitions labeled by a guard are complentary and exclusive. The actions may be *elementary actions* or *communication actions*.

- An elementary action (an assignment for example) does not involve other services; it does not use a communication channel.
- A communication action is either a *service call/response* or a message *communication*. Service call/response is treated as a communication. Therefore communications are matching pairs: *send(!)-receive(?), call service(!!)-wait service start(??), emit service result(!!)-wait service result(??)*. The Kmelia syntax of a communication action (inspired by the Hoare's CSP) is: `channel(!|?|!!|??) message(param*)`.

[1] We currently implemented sequential combinations of actions but other operators are on the way (parallel, choice, ...).

- A *branching transition* is a (sub-)services of the component C that should be called when the evolution reaches this transition. This provides description flexibility in collapsing and sharing nested service descriptions. Branching transitions are related to branching states.

Channels
Message communications and external service calls use a channel that is established between services when assembling components. A service call refers to an internal service of the same component or an external service required by the component. An internal service is a sub-service or a provided service of the component interface. At the moment one writes a behaviour, one does not know which components will communicate, but one has to know which channel will be used. The channel is usually named after the required service that represents the context. The placeholder keyword `CALLER` is a special channel that stands for the channel opened for a service call. From the point of view of a provided service p, `CALLER` is the channel that is open when p is called. From the point of view of the service that calls p, this channel is named after one of its required service, which is probably named p. The placeholder keyword `SELF` is a special channel that stands for the channel of an internal service call.

2.2 Component Specification

A component (C) is a 8-tuple $\langle \mathcal{W}, Init, \mathcal{A}, \mathcal{N}, I, \mathcal{D}_S, \nu, \mathcal{C}_S \rangle$ with:

- $\mathcal{W} = \langle T, V, V_T, Inv \rangle$ the state space where
  - T is a set of predefined types,
  - V a set of variables,
  - $V_T \subseteq V \times T$ a set of typed variables,
  - Inv is the state invariant;
- $Init$ the V_T variable initialisation;
- $\mathcal{A}$ a finite set of elementary actions;
- $\mathcal{N}$ a finite set of service names;
- I the component interface which is the union of two disjoints finite sets: I_p (resp. I_r) the set of names of the provided (resp. required) services that are visible in the component environment.
- $\mathcal{D}_S$ is the set of service descriptions which is partitioned into the provided services ($\mathcal{D}_{S_p}$) and the required services ($\mathcal{D}_{S_r}$).
- $\nu : Names \rightarrow \mathcal{D}_S$ is the function that maps service names to service descriptions. Moreover there is a projection of the I partition on its image by ν:
  $n \in I_p \Rightarrow \nu(n) \in \mathcal{D}_{S_p} \wedge n \in I_r \Rightarrow \nu(n) \in \mathcal{D}_{S_r}$
- $\mathcal{C}_S$ is a predicate related to the services of the interface of $\mathcal{C}$ in order to constrain (or control) the usage of the services.

The state of the component is defined by its variable and its invariant. The invariant is a predicate on the state variables and global definitions (constants, variables, functions). This aspect is not described in the current report but it relies on state-based formal models like Z [30] or B [1].

According to our model the behaviour of the component relies on the behaviours of its services. When it is needed in a specific application, $\mathcal{C}_{S_p}$ is used to describe conditions on the service usage: it may be either an ordering of services (a *Component Behaviour Protocol* in the sense of of [17, 27]) or a logic predicate (to ensure properties like mutual exclusion). Optionally, behaviours of services can be used to describe component protocols using a simple extension of the eLTS which is not described here.

2.3 Links, Assembly and Composition of Components

In the Kmelia model, the component assembly and the component composition are based on various types of links between services. For instance we have a sublink when a hidden service is called in the scope of a provided service. In an assembly, required services are linked to provided services. A composition is an assembly where some unlinked services are promoted to the composite level. In this section, we provide the formal background for component assembly and composition.

We use in the following a set theory notation close to that of Z or B where $\mathbb{P}\ X$ denotes the powerset of X (all its subsets), $X \leftrightarrow Y$ denotes the relation from X to Y (a set of couples), $(\forall D \bullet P)$ is a predicate P with declared types variables $D = x : T$, id denotes the identity relation; dom and ran denote respectively the domain and the range of a relation; $a \mapsto b$ denotes the couple (a, b).

Let $\mathcal{C}$ be a set of C_k components with $k \in 1..n$ and
$C_k = \langle\langle T_k, V_k, V_{Tk}, Inv_k\rangle, Init_k, \mathcal{A}_k, \mathcal{N}_k, I_k, \mathcal{D}_{Sk}, \nu_k, \mathcal{C}_{Sk}\rangle$.

2.3.1 Dependencies between Component Services

Let $depends_k$ a relation between component services defined as a part of the service dependency in a component C_k:

$$depends_k : \mathcal{N}_k \leftrightarrow \mathcal{N}_k$$
$$\forall (n, m) : depends_k \bullet (n \in cal_{sm}) \vee (n \in req_{sm}) \vee (n \in sub_{sm})$$

where $sm = \nu_k(m)$.

2.3.2 Links and Sublinks between Component Services

Let $\mathcal{N}$ be a set of service names of $\mathcal{C}$: $\mathcal{N} = \bigcup_{k \in 1..n} \mathcal{N}_k$.

Basically, links are 4-tuple of component and service names with the following property: (1) the service names are those of their component, (2) any component service is not linked to itself.

To simplify the presentation we take some freedom with the notation, for instance the quantification over the tuples. This is to be considered as a draft (shorthand) notation.

$$BaseLink : \mathbb{P}\ (\mathcal{C} \times \mathcal{N} \times \mathcal{C} \times \mathcal{N})$$
$$(1) \quad \forall (C_i, n_1, C_j, n_2) : BaseLink \bullet n_1 \in \mathcal{N}_i \wedge n_2 \in \mathcal{N}_j$$
$$(2) \quad \forall C_i : \mathcal{C},\ n_1 : \mathcal{N}_i \bullet (C_i, n_1, C_i, n_1) \notin BaseLink$$

A link is a basic link over two services of the interface of the components.

$$Link \subseteq BaseLink \wedge$$
$$\forall (C_i, n_1, C_j, n_2) : Link \bullet n_1 \in I_i \wedge n_2 \in I_j$$

A sublink is a basic link over two services, one of them at least is not in the interface of the components.

$$SubLink \subseteq BaseLink \wedge$$
$$\forall (C_i, n_1, C_j, n_2) : SubLink \bullet n_1 \notin I_i \vee n_2 \notin I_j$$

The *Sublink* set makes explicit the relation between the services dependencies declared in the interfaces of the services concerned by a *Link*. In the following these relations are constrained in order to define a specific component assembly and component composition.

2.3.3 Component Assembly

Assembling components consists in linking pairwise services. A **component assembly** is a triple $A = (\mathcal{C},\ links,\ subs)$ where $\mathcal{C}$ is a set of components, $links$ is a set of links between the services of $\mathcal{C}$ and $subs$ is a relation from links to sublinks.

$$
\begin{array}{ll}
& links \subseteq Link \land \\
(1) & \quad (\forall (C_i, n_1, C_j, n_2) : links \bullet C_i \in \mathcal{C} \land C_j \in \mathcal{C} \land \\
& \qquad ((n_1 \in I_{p_i} \land n_2 \in I_{r_j}) \lor (n_1 \in I_{r_i} \land n_2 \in I_{p_j}))) \\
& subs : Link \leftrightarrow SubLink \\
(2) & \quad \operatorname{dom} subs = links \land \\
(3) & \quad (\forall ((C_i, n_1, C_j, n_2) \mapsto (C_k, n_3, C_l, n_4)) \in subs \bullet C_i = C_k \land C_j = C_l) \land \\
(4) & \quad (\forall (C_i, n_1, C_j, n_2) : \operatorname{ran} subs \bullet ((\nu_i(n_1) \in \mathcal{D}_{Sp_i}) \operatorname{xor} (\nu_j(n_2) \in \mathcal{D}_{Sp_j})))
\end{array}
$$

The components of the links are the components of the assembly (1). The sublinks are related to links (2) that concern the same components (3). Provided services are linked to required services (1 and 4).

A is a *well-formed component assembly* if the following property holds: the services in the sublinks are not from their component's interface, they are a dependency of the service of their baselink (w.r.t $sublinks$).

$$
\begin{array}{ll}
(5) & \quad \forall ((C_i, n_1, C_j, n_2) \mapsto (C_k, n_3, C_l, n_4)) \in subs \bullet \\
& \qquad ((n_3 \mapsto n_1) \in depends^* \lor (n_4 \mapsto n_2) \in depends^*)
\end{array}
$$

where $depends^*$ is the transitive closure of $depends$.

Practically a *link* establishes an implicit communication channel between the involved services. This channel is also used for the communication between the related sub-services.

Restrictions
The basic component model presented above is restricted with following constraints: a component is both a component type and the unique instance of it, a required service can be linked to at most one provided service (no overloading), single instantiation of a service at any time. This frame has to be extended later to handle the following cases: multiple clients, various providers, broadcast communications, etc.

2.3.4 Component Composition

A *composition* is a well-formed component assembly which is encapsulated within a component. We define an operator named compose that builds a new component by combining one or several components.
The parameters of the compose operator are:

- an outer component oC (the composite) together with its interface, new services and services of its constituents;
- a well-formed assembly $A = (\mathcal{C},\ links,\ subs)$ (see section 2.3.3);
- the desired $promotions$, they are set of links between the services of oC and those of $C_k \in \mathcal{C}$.

The promotion is a relation between a service of the composite oC and an unlinked service of the components in A, that preserves existing sublinks; such promoted service becomes usable at the composite level.

$$
\begin{array}{ll}
& promotions \subseteq BaseLink \land \\
& \quad (\forall (C_i, n_1, C_j, n_2) : promotions \bullet \\
(1) & \qquad (C_i = oC) \land (C_j \in \mathcal{C}) \land \\
(2) & \qquad ((\nu_{oC}(n_1) \in \mathcal{D}_{Sp_{oC}} \land n_2 \in I_{p_j}) \lor (\nu_{oC}(n_1) \in \mathcal{D}_{Sr_{oC}} \land n_2 \in I_{r_j})))
\end{array}
$$

The resulting component is an enhancement of oC: it contains every provided and required services of oC and provides and requires the services that were promoted from other components in $\mathcal{C}$ by using $promotions$. We consider here that sub-services of the promoted services are also promoted.

From the methodological point of view, the composition operator may be used to refine an abstract component with a component assembly; it may also be used to structure simple components or to provide a more restrictive interface of an existing component.

3 A Case Study

In this section, we apply our component model to a real-world case, a bank automatic teller machine (ATM). Since the problem is well known, only a reduced version its statement is presented below[2]. Hardware and device issues are out of the scope of the current study.

3.1 Problem Statement

An ATM provides several bank services to customers (withdrawal, deposit, transfer, query) and achieves maintenance and security services. Among them, we focus on the cash dispenser and query services. The query service is only valid for the (local) bank customer. Both services require the same identification step. The ATM asks the card holder (user) to insert a card. The card holder introduces the withdrawal (or cash) card in the ATM. The ATM accepts the card and reads its serial number (card identifier). If the card is readable, the ATM requests the user password otherwise it rejects the card. The user enters the password. The ATM verifies the given password (compared with the card password). If the verification succeeds, the ATM authenticates the card holder, otherwise the ATM requests the password again. When the verification fails three times (the number of trials is closely related to the identification procedure), the ATM swallows the card.

After the card holder identification in the withdrawal service, the ATM requires an authorization from its ACD/ATM controller (AAC), that represents the bank management. If the AAC accepts the transaction, the ATM asks for the amount of cash, otherwise the card is ejected and the withdrawal transaction ends. The user enters an amount which is compared with the current card policy limit. If the allowed amount is lower than the requested or if the current ATM cash is not sufficient, the ATM asks for the amount of cash again. Otherwise, the ATM asks the AAC to process the transaction, updates the card limit, dispenses the cash and try to print a receipt if it is requested. In any case the withdrawal transaction ends after a card ejection.

As fas as the query service is concerned, after the card holder identification, the ATM checks that the card is related to a local bank account. If the card holder is not a local bank customer the transaction stops and the card is ejected. Otherwise, the ATM asks the kind of query (last transactions, account balance) to the user and processes it. Several queries are allowed. The transaction ends with a special "quit" query. Then the card is ejected and the query transaction ends.

Remember that many aspects (such as screen messages, device operations) are omitted in the above statement.

3.2 An Architecture Overview: the Component Assembly

A component model is the description of the individual components and the component assembly. From a methodological point of view, both top-down and bottom-up approach exist. For this case study, we follow a top-down approach.

Figure 1 shows a simplified component assembly for the ATM. The ATM includes four components: the central `ATM_CORE` that handles the ATM bank services, the `USER_INTERFACE` component controls the user access, the `AAC` stands for the bank management and the `LOCAL_BANK` handles the bank management access. Remember again that we strongly simplified the model because we only want to illustrate our model and focus on the withdrawal transactions. For example, `AAC` and `LOCAL_BANK` could be the same component.

[2] A detailed version of the problem statement can be found in chapter 8 of [28] or in http://www.commoncriteriaportal.org/public/fi les/ppfi les/PP9907.pdf

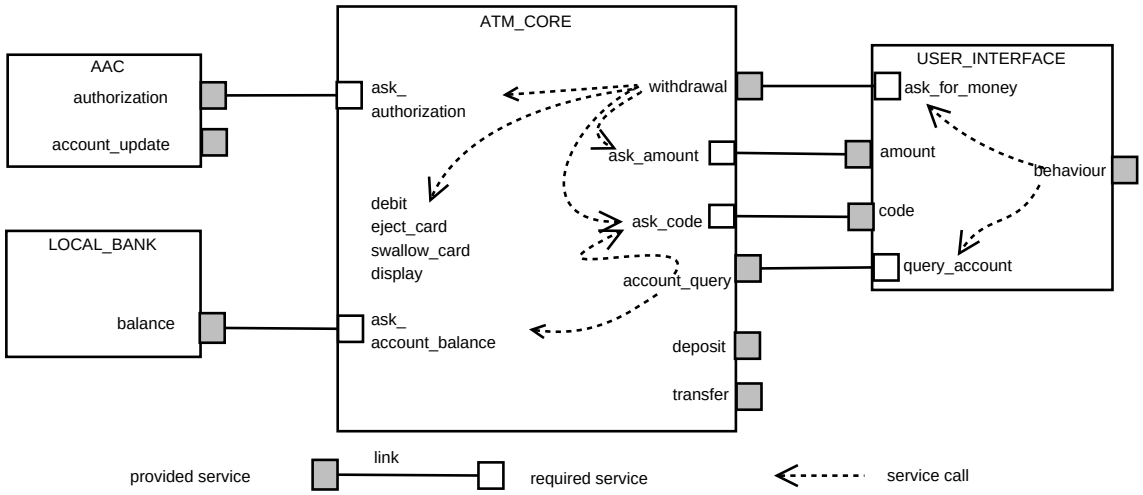

Figure 1 : *Assembly for an ATM System*

The following naming conventions hold: component names are uppercase nouns, provided services are lowercase names, required services are lowercase verbal phrases.

In an assembly, required services are linked to provided services. For example, the `ask_for_money` required service is fulfilled by the provided `withdrawal` service. Reciprocally, a provided service calls its own required services to delegate some processing. For example, the `withdrawal` provided service calls the required `ask_code` and `ask_authorization` services to check the card holder's rights. Note that the `ask_code` is a sub-service because it is not in the provided interface of `ATM_CORE` (see the textual version).

In this example, every required service is linked but this is note a rule because one can use partially a component. The component usage is quite flexible: an assembly may be valid for one service only, since its dependency chain is fulfilled, the sub-services are invoked optionally thus the `ask_for_money` service may operate with any withdrawal protocol (whatever the order for amount and code). A non linked required service prevents the usage of all the provided services that call it, directly or not. Similarly, unlinked provided services, *e.g* `withdrawal` are free of use. They can be linked at an upper level, if the assembly is encapsulated in a composition.

The *USER_INTERFACE* component offers the (provided) `code` service only in the interface of the `behaviour` service; it means that the *USER_INTERFACE* only gives its code during a withdrawal operation that it has initiated. In such a situation, `code` is a sub-service. The component services are detailed in the Fig. 2. Note that the *USER_INTERFACE* may also call a `withdrawal` service that does not require its `code`.

3.3 Specification of a Component

The component specification is the description of the state space, the interface and the services (see Section 2.2). In a component definition there are no assumption on the component (type) that can fulfill the required services or request the provided services. The following Kmelia source specifies the component `ATM_CORE` of figure 1.

```
# Specification of the ATM_CORE component
# Author: Pascal Andre
# Date: 11/07/05
#
# Note : Only the features of interest for the withdrawal service
#        are included in the current specification.
#
COMPONENT ATM_CORE
#
# The ATM_CORE component is the central component for a bank cashier station.
# The main services of such a system are cash withdrawal, account query, deposit money
# and transfer bank query.
# The current specification focuses only on cash withdrawal.
```

```
#
INTERFACE
  provides : {withdrawal, account_query, deposit, transfer}
  requires : {ask_authorization, ask_account_balance}
TYPES
  CashCard  :  {code:Integer, id:Integer, limit:Integer} # record type
CONSTANTS
  available_cash : Integer := 100,
  swallowed_size : Integer := 100
VARIABLES
  name : String,
  swallowed_cards : Set,
  available_notes : Integer
PROPERTIES
 cash_disp: available_notes >= 0,
 card_capacity: size(swallowed_cards) <= swallowed_size
INITIALIZATION
  name := "ATM203";
  swallowed_cards := emptySet;
  available_notes := 10000;

SERVICES
# services from external files (currently only in the same directory) can be included
provided external withdrawal
provided external account_query
provided external deposit
provided external transfer

#a fool service for testing dependencies
provided ident ()
 Interface
   # subprovides : {}
   # calrequires : {}
   # extrequires : {}
Behavior
init i # i is the initial state
final f # i is a final state
{
  i - display("my id") -> e1,
  e1 - __CALLER!!ident() -> f
}
end

#required services
required ask_authorization (id : Integer, code : Integer) : Boolean
end
required ask_account_balance (id : Integer) : Integer
end

#internal services
internal debit (c : CashCard, m : Integer)
end
internal eject_card()
end
internal swallow_card()
end
internal display(msg : String)
end

END_SERVICES

# end of ATM_CORE
```

Since component models get complete along the development process, the component specification can also be seen at several abstraction levels. For instance, at a high abtraction level, the interface and its services may

outweigh state considerations and state variables can be optional or limited, handled by internal actions. This is the case in our current specification: we mainly focus on services.

In its state space, the ATM_CORE state includes three variables

1. name, which is the name of the ATM,
2. swallowed_cards, which is a collection of swallowed cards,
3. available_notes, which is the current amount of available notes,

and two constants

1. available_cash, which is the amount of cash necessary to start a withdrawal,
2. swallowed_size, which is the number of cards that the swallow area can contain.

The invariant asserts that the swallow area has a limited size and the available cash is positive.

There are four internal services (debit, eject_card, swallow_card, display). These represent ongoing services definitions. In a top-down modelling approach, ongoing services represent those services that will be described later in a deeper design. For example, the *debit* action can be refined by calling the AAC update_account service and the CASH_DISPENSER (a new component) dispense_cash service (see Section 3.5).

All the usage constraints are associated to services and the constraints among services are set in $\mathcal{C}_S$. These constraintes are twofold :

1. Ordering constraints: the provided services can be called in a specific order. This corresponds to a general component behaviour. We simply model the order as a special service. For example, the behaviour main service defines the body of USER_INTERFACE.
2. Applicability (dynamic) constraints: two provided services may be exclusive, or two required services should be supplied by the same component...

We did not consider service constraints for the ATM_CORE component.

3.4 Specification of Services

The provided withdrawal service of the component ATM_CORE of figure 1 is specified as follows:

```
# Specification of the  component
# Author: Pascal Andre
# Date: 11/07/05
#
# Component: ATM_CORE
#
#This is a normal provided service: withdrawal

withdrawal (card : CashCard)
Interface
   subprovides : {ident}
   calrequires : {ask_code, ask_amount} #required from the caller
   extrequires : {ask_authorization}

Pre
  #service available if there is enough money
  available_notes >= available_cash

Variables # local to the service
  nbt : Integer,    # nbt : number of authorized trials of code entering
  c : Integer,      # c : input code given by the user
  a : Integer,      # a : input amount given by the user
```

```
  rep : Boolean,    # rep : reply from the authorization request
  success : Boolean # success : result of the withdrawal request

Behavior
init i # i is the initial state
final f # i is a final state
{
  i - {
      nbt:= 3 ;
      # set the number of authorized trials
      success := false
      # by default the withdrawal fails
  } -> e0,
  e0 - __CALLER!!ask_code() -> e1,
  # call the required service ask_code of the caller (implicit)
  e1 - {
      __CALLER??ask_code(c) ;
      # input communication: gets the password on the ask_code (service) channel
      nbt := nbt -1
      # the number of trials decreases
  } -> e2i,
  e2i - __CALLER!rdv() -> e2,
  e2 - [c=card.code] rep := _ask_authorization!!ask_authorization(card.id,c) -> e3,
  # call the required service ask_authorization on the channel ask_authorization
  e2 - [c<>card.code && nbt>0] display("Enter your card code, please ")-> e0,
      # call an internal action
  e2 - [c<>card.code && nbt=0] {
      display("Card swallowed, sorry");
      # call an internal action
      swallow_card()
      # call an internal action
  } -> e4,
  e3 - [rep] display("Enter the cash amount, please ?") -> e5,
  # the AAC accepts the transaction, the amount is asked
  e3 - [not rep] {
      # the AAC refuses the transaction, the service ends
      display("Transaction refused") ;
      # call an internal action
      eject_card()
      # call an internal action
  } -> e4,
  e4 - __CALLER!!withdrawal(false) -> f,
  # the withdrawal fails, the caller is informed from the result on the
  # withdrawal service channel. The service ends.
  e5 - __CALLER!!ask_amount() -> e6,
  # call the required service ask_amount of the caller (explicit)
  e6 - __CALLER??ask_amount(a) -> e7,
  #  input communication: gets the password on the ask_amount (service) channel
  e7 - [m<= card.limit] {
      debit(c,m);
      # call an internal action
      eject_card()
      # call an internal action
  } -> e8,
  e7 - [m > card.limit] display("require too much money, please enter the amount again") -> e3,
  # call an internal action
  e8 - {
      success := true ;
      # the withdrawal succeeds
      __CALLER!!withdrawal(success)
      # the caller is informed from the result on the
      # withdrawal service channel. The service ends.
  } -> f
}
Post
```

```
  available_notes >= pre(available_notes)
  # (success && (available_notes = pre(available_notes) - m)) ||
  # ((not success) && available_notes = pre(available_notes))
end
```

The withdrawal starts by an identification step: card insertion, password control, authentication by ACD/ATM controller (AAC). If the AAC accepts the transaction, the ATM asks for the amount of cash, otherwise the card is ejected and the withdrawal transaction ends. The user enters an amount which is compared with the current card policy limit. When the allowed amount is lower than the requested one or if the current ATM cash is not sufficient, the ATM asks again for the amount of cash. Otherwise the ATM asks the AAC to process the transaction, updates the card limit, gives the cash and prints a receipt when it is possible. In any case the withdrawal transaction ends after a card ejection. There are four internal service call (*debit*, *eject_card*, *swallow_card*, *display*). The channels can be omitted and deduced either from the context or from default rules. This syntactic sugar is not currently implemented in our prototype.

3.4.1 Specification of the Sub-Services

The `USER_INTERFACE` component (figure 2) is quite flexible ; it can operate with any withdrawal protocol whatever order and what number is given for amount and code. IN other words, the interaction description is made very flexible by enabling the calls of sub-services when the evolution reaches given states. The sub-services available in one state are listed between angle brackets and annotate the (branching) state. The notation `e1 <code, amount>` expresses that the services `code` and `amount` of the `USER_INTERFACE` component may be called in the `e1` state. The behaviour description remains simple (the LTS is not huge) since the descriptions of the sub-services are deferred.

```
# Specification of the USER_INTERFACE component
# Author: Pascal Andre
# Date: 11/07/05
#
# Note : Only the features of interest for the ask_for_money required service
#        are included in the current specification.
#
COMPONENT USER_INTERFACE
#
# The USER_INTERFACE component is a client component for a bank cashier station.
# The main services of such a system are cash withdrawal, account query, deposit money
# and transfer bank query.
# The current specification focuses only on the client behavior for a cash withdrawal.
#
INTERFACE
  provides : {behavior, amount}
  requires : {ask_for_money, query_account}

TYPES
  CashCard  :  {code:Integer, id:Integer, limit:Integer} # record type

VARIABLES
  myCard : Card,    # user card
  myCode : Integer  # user code

SERVICES
#This is a client main service: behavior

provided behavior ()
# specified as an infinite service: main loop
Interface
   subprovides : {code} #, amount
   #calrequires : {} #required from the caller
   extrequires : {ask_for_money, query_account}
Variables # local to the service
```

```
  b : Integer       # balance of my account
Behavior
init e0 # e0 is the initial state
final e0 # e0 is a final state (loop behavior)
{
  e0 - {
      display("Hello, please insert your card");
      read(myCard)
      #internal I/O actions
  } -> e0,
  e0 - _ask_for_money!!ask_for_money(myCard) -> e1,
  # invocation of the withdrawal service
  e0 - _query_account!!query_account(myCard) -> e10,
  # invocation of the query_account service
  e1 <code>, # specifies callable subservices on node e1 , amount
  e1 - _ask_for_money?rdv() -> e2,
  e2 - _ask_for_money??ask_for_money(myCard) -> e0,
  # wait for the result of of the ask_for_money service

  e10 <code>, # specifies callable subservices on node e10
  # unspecified yet after e10
  e10 - _query_account??query_account(b) -> e0
}
end

provided code () : Integer
# answers the code of the user
Interface
   #subprovides : {}
   calrequires : {getId} #required from the caller
   #extrequires : {}
Variables # local to the service
  id : Integer       # bank id
Behavior
init e0 # e0 is the initial state
final f # f is a final state
{
  e0 - {
      display("Please enter your card code");
      read(myCode) # component variable
  } -> e1,
  e1 - __CALLER!!getId() -> e2, #__CALLER
  e2 - __CALLER??getId(id) -> e3,
  e3 - store(id, today) -> e4,
  e4 - __CALLER!!code(myCode) -> f
  # send the code
}
end

provided amount () : Integer
# answers the amount of cash that the user wants
Variables # local to the service
  a : Integer       # amount of cash wanted
Behavior
init e0 # e0 is the initial state
final e0 # e2 is a final state
{
  e0 - {
      display("Please enter the amount of cash");
      read(a) # local variable
  } -> e1,
  e1 - __CALLER!!amount(a) -> e2
  # send the amount of cash
}
end
```

```
#required services
required ask_for_money (card : CashCard) : Boolean
Interface
   subprovides : {code}
end
required query_account (card : CashCard) : Integer
end
END_SERVICES

# end of USER_INTERFACE
```

The user (card holder) behaviour is simply to ask for money or query a (local) account ; these are required services from the user's point of view. The ATM-user interaction starts by the call of `ask_for_money` in `USER_INTERFACE`. Then `amount` and `code` become available services, in the context of the `e1` state.

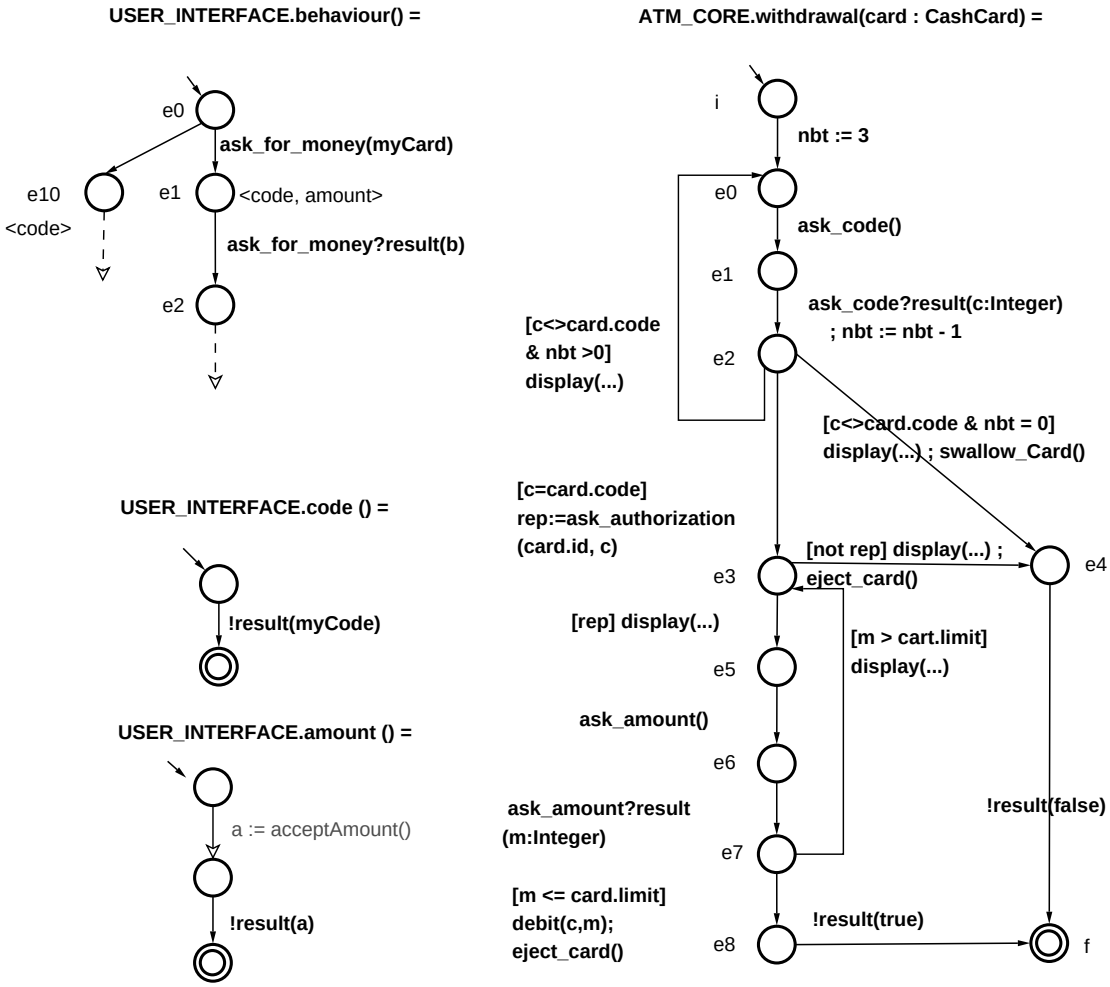

Figure 2 : *Behaviours of Component Services*

The above specifications have a visual representation which is more user-friendly. Figure 2 is a partial visualization of the services of interest (for a verification). The behaviour of a service is a finite state machine. The sub-services available in one state are quoted by state labels using angle brackets:
the notation `e1 <ask_code>,<ask_amount>` means that these services can be called in that state only. In

practice, this shortens the graph by collapsing ring sequences of transitions. Bloc transitions are those transitions in which labels include the sequential ";" operator, which is also a notation abbreviation.

3.5 Compositions for the ATM

The composition operator is used to define component assemblies (verification goal) or to refine specifications (structuration).

Assembly Composition In Kmelia, an assembly is specified using the composition operator. For example, let define a component that includes only the USER_INTERFACE and ATM_CORE components.

```
// Specification of the ATM_SYSTEM composite component
// Author: Pascal Andre
// Date: 11/07/05
//
// Note : Only the features of interest for the withdrawal service
//        are included in the current specification.
//
COMPONENT ATM_SYSTEM
//
// The ATM_SYSTEM component represents an assembly with the ATM_CORE and USER_INTERFACE components
// the unsolved required services are the system required services
// the unsolved provided services are the system provided services
//
INTERFACE
  provides : {behavior}
  requires : {ask_authorization, ask_account_balance} //loaded but unused

SERVICES
END_SERVICES

// -------------
// the contents of the "assembly"
// -------------

COMPOSITION
// list of components included in ATM_SYSTEM
// the following line looks for ATM_CORE.cmp and USER_INTERFACE.cmp
{ATM_CORE USER_INTERFACE},

// list of links between services
{
(p-r ATM_CORE.withdrawal, USER_INTERFACE.ask_for_money
 //sublinks
 (r-p ATM_CORE.ask_code, USER_INTERFACE.code)
 (r-p ATM_CORE.ask_amount, USER_INTERFACE.amount)
 //TODO subsublink
 (p-r ATM_CORE.ident, USER_INTERFACE.getId)
)
// the service withdrawal of the ATM_CORE is connected to
// the service ask_for_money required by USER_INTERFACE

(p-r ATM_CORE.account_query, USER_INTERFACE.query_account
 //sublinks
 (r-p ATM_CORE.ask_code, USER_INTERFACE.code)
 //TODO subsublink
 (r-p USER_INTERFACE.getId, ATM_CORE.ident)
)
// the service account_query of the ATM_CORE is connected to
// the service query_account required by USER_INTERFACE

(p-p USER_INTERFACE.behavior, SELF.behavior)
// the provided service behavior is connected to the provided service behavior of USER_INTERFACE
```

```
   // this line defines the behavior of behavior unless specified (which is not the case here),
   // the interface of behavior is the same as USER_INTERFACE.behavior
   // this link use the optional prefix (p-p)

   (r-r ATM_CORE.ask_authorization, SELF.ask_authorization)
   // the required service ask_authorization is connected to the required service ask_authorization
of ATM_CORE
   // this link use the optional prefix (r-r)

   (r-r ATM_CORE.ask_account_balance, SELF.ask_account_balance)
   // the required service ask_account_balanceis connected to the required service ask_account_balance
of ATM_CORE
   // this link use the optional prefix (r-r)

   // -------------
   // incompleteness of the "assembly"
   // -------------
   // The following provided services are left unsatisfied
   // ATM_CORE.deposit
   // ATM_CORE.transfer
   // Indeed, they are not useful for the target assembly.

   }
```

This approach provides a means for verifying the composability of assemblies. In the following we focus on the `withdrawal` provided service which is linked to the required `ask_for_money` service, called by the `behaviour` service. This triple constitutes a *service context* for a service verification (see section 4.2.1).

Structuration Composition The composition operator is used to structure component specifications (bottom-up approach) or to refine specifications (top-down approach). In the ATM case study, we refine the current specification of assembly figure 1 in delimiting the context for an ATM system. In particular, we make some internal actions explicit. The `display` internal action is delegated to a `SCREEN` component. The `swallow_card` and `eject_card` internal actions are delegated to a `CARD_DEVICE` component. The `debit` internal action is delegated both to an internal `CASH_DISPENSER` component and the `update_account` required service. The `display` internal action is delegated to a `SCREEN` component. The composition is presented in figure 3.

The `CASH_DISPENSER`, `CARD_DEVICE` et `SCREEN` components are defined as follows.

```
Component CASH_DISPENSER
interface
  provided : {cash_dispense, ...}
  required : {}
constants
  available_cash : Integer = 100 ;
  ...
variables
  available_notes : Dictionary ;  ...
invariant
   available_notes := empty;
  ...
initialisation
   available_notes.sum >= 0
  ...
services
  cash_dispense : (amount : Integer) =
  PRE   (amount > 0) &
        (available_notes.sum  >= amount)
  SPEC
     {e0 - ...
 - available_notes :=
         available_notes.dispense(montant) ...
     eX - ... --> f
     }
  POST  old(available_notes).sum -
        (available_notes.sum)  - amount = 0
     // no holes
end
```

```
Component CARD_DEVICE
interface
  provided : {card_swallowing, card_ejection}
  required : {}
services
  card_swallowing = ...
  card_ejection = ...
end
Component SCREEN
interface
  offerts : {display}
  requis : {}
services
  display (m:String)= ...
end
```

Now, the ATM component is defined using the compose operator and the components ATM_CORE, CASH_DISPENSER, CARD_DEVICE et SCREEN.

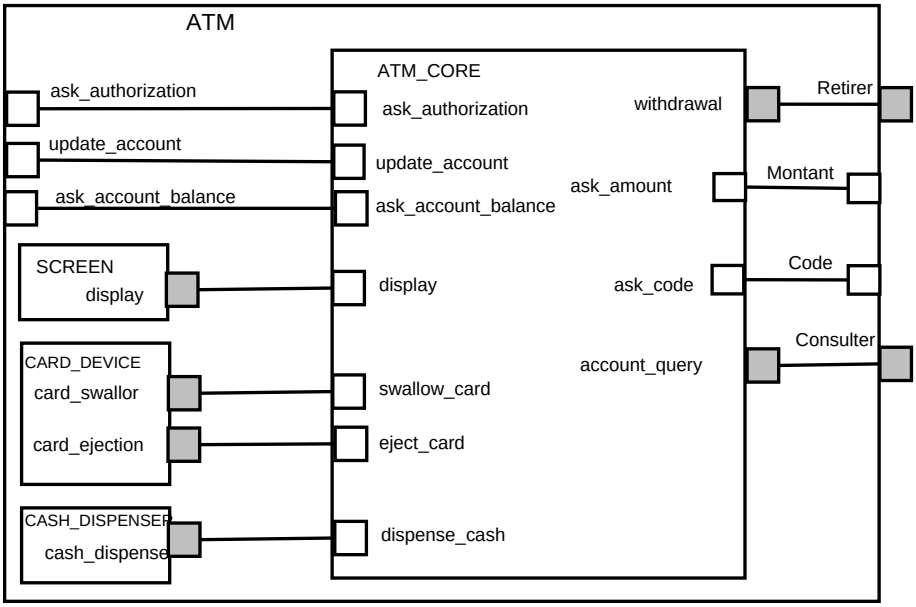

Figure 3 : *The new ATM is a composite component*

4 Formal Verification of Components and Assemblies

The specifications of components, compositions and assemblies should be formally analysed to ensure the quality of the developped systems. The formal analysis may be performed according to various aspects. First, in section 4.1 we overview the main issues of component analysis (formal verification) and certification. Second, we focus on one specific property: the *composability* of components and its verification. The composability is the correct interaction between components. Its definition is given in section 4.2. This definition covers a wide range of properties, thus its verification is quite complex.

In the remaining of the report, we address the problem of the verification of the *behavioural compatibility* of component assemblies. We explore several ways for implementing a verification rules: in the section 4.5 we present a verification algorithm; then in the section 5 and 6 we reuse existing verification tools or environments.

4.1 Formal Analysis Aspects

There are several aspects on which verification may be conducted. Specifically, verification aspects of components include:

- *(Static) Interoperability properties*: compatibility of signatures and interfaces (naming and typing); do a component gives enough information about its interface(s) in order to be (re)usable by other components;
- *Architectural properties*: that means the availability of the required components, the availability of the needed services (completeness), the correctness of the links between interfaces of components (providers and callers);
- *Behavioural compatibility*: it is about the correct interaction between two or more components which are combined. Several points need to be considered: various kinds of interaction, synchronous or not, atomic actions or non atomic ones.
- *Correctness of functional properties*: do the components do what they must do? These properties may be checked independently on the components which are used and also on the composition of the components;

- *Flexibility of maintenance (modifiability, evolution)*: that means the components should be simply updated on needs, without affecting drastically the third party components which use it. The update of a component includes, the modification of the implementation of its service(s), the remove/adding of a service, etc.
- *Heterogeneity*: within the CBSE approach, the components coming from various providers may be composable to develop large systems. This is a challenging concern because the components may have different models.
- *Compositionality* is also an important concern: the properties of a global system should be inferred from the properties of the composed components.

In the following we study a property which is central to any combination of components: the *composability* of components in an assembly. This is a quite complex property that covers static and dynamic aspects of the specification. Hence, the first three categories of properties are related to composability.

4.2 Composability

A property to be proved needs first to be formally expressed. In this section, we define the correctness of a component assembly, the **composability** property. The links are central here (since they are used for assemblies). A link defines the context of a service composability. Therefore, the composability is basically defined on services and generalized to the components. In the following, both static aspect (interface) and dynamic aspect (behaviour) are considered to check composability.

Definition 4.1 (Service Context)
*Let $A = (\mathcal{C},\ links,\ subs)$ be a component assembly, and C_i and C_j two components of A ($C_i \in \mathcal{C} \wedge C_j \in \mathcal{C}$). A **service context** is a triple $SC_A = (sp_{C_i}, sr_{C_j}, sp_{C_j})$ such that*

1. *sp_{C_i} is a provided service of the component C_i ($sp_{C_i} \in \mathcal{D}_{Sp_i}$) with $sp_{C_i} = \langle\langle \sigma_p, P_p, Q_p, V_{sp}, S_{sp}\rangle, \mathcal{B}_{sp}\rangle$,*
2. *sr_{C_j} is a required service of a component C_j ($sr_{C_j} \in \mathcal{D}_{Srj}$) with $sr_{C_j} = \langle\langle \sigma_r, P_r, Q_r, V_{sr}, S_{sr}\rangle, \mathcal{B}_{sr}\rangle$,*
3. *sp_{C_j} is a provided service of a component C_j ($sp_{C_j} \in \mathcal{D}_{Sp_j}$),*
4. *there is a link between sr_{C_j} and sp_{C_i} in A: $\exists (C_i, n_1, C_j, n_2) \in links \bullet$ $(\nu_i(n_1) = sp_{C_i} \wedge \nu_j(n_2) = sr_{C_j}) \vee (\nu_j(n_2) = sp_{C_i} \wedge \nu_i(n_1) = sr_{C_j})$*
5. *sr_{C_j} is required by sp_{C_j} ($sr_{C_j} \in req_{sp_{C_j}}$).*

The service context is a basis for the definition of service composability. The service sp_{C_i} is called the **reference service**.

4.2.1 Service Composability

The service composability consists to check that a provided service can be fullfilled by a provided service w.r.t a link: their interfaces are compatible and their behaviour are compatible through the link (for each provided service that calls the required service the interactions between two provided services do not lead to errors).

Definition 4.2 (Service Composability)
Let $A = (\mathcal{C},\ links,\ subs)$ be a component assembly, and C_i and C_j two components of A ($C_i \in \mathcal{C} \wedge C_j \in \mathcal{C}$). A provided service $sp_{C_i} = \langle\langle \sigma_p, P_p, Q_p, V_{sp}, S_{sp}\rangle, \mathcal{B}_{sp}\rangle$ of a component C_i and a required service $sr_{C_j} = \langle\langle \sigma_r, P_r, Q_r, V_{sr}, S_{sr}\rangle, \mathcal{B}_{sr}\rangle$ of a component C_j are s-composable (noted s-composable(sp_{C_i}, sr_{C_j})) w.r.t a link $(C_i, n_1, C_j, n_2) \in links$ (or a link $(C_j, n_2, C_i, n_1) \in links$) when n_1 is the name of sp_{C_i} ($\nu_i(n_1) = sp_{C_i}$), n_2 is the name of sr_{C_j} ($\nu_j(n_2) = sr_{C_j}$ and sr_{C_j} is required in at least one provided service s of C_j ($sr_{C_j} \in req_{sp_{C_j}}$), if:

1. *the interfaces of sp_{C_i} and sr_{C_j} are compatible; that is,*
   (a) *their signatures are matching (no type conflict: σ_p and σ_r are type compatibles[3]),*
   (b) *the assertions (pre/postconditions) are consistent*
   $(pre(sr_{C_j}) \Rightarrow pre(sp_{C_i}) \wedge post(sp_{C_i}) \Rightarrow post(sr_{C_j}))$ *and*
   (c) *their service dependency are (deeply) compatible: the inner required-provided relationship is preserved (recursion on interface compatibility including the sublinks of A, w.r.t $subs$).*
2. *for each service context $SC_A = (sp_{C_i}, sr_{C_j}, sp_{C_j})$ where $sr_{C_j} \in req_{sp_{C_j}}$, the behaviour of sp_{C_i} and sp_{C_j} are compatible: $compatible(\mathcal{B}_{sp_{C_i}p}, \mathcal{B}_{sp_{C_j}})$; that is, their eLTSs are matching; either they evolve independently or they perform complementary communication actions until a termination without a deadlock.*

We currently treat the interface compatibility by a static analysis, without checking property 1.b. The compatibility of behaviours is dealt with in more details in the remaining of the report, using mechanized techniques.

4.2.2 Component Composability

The composability of two (or more) components is a generalization of the composability of their linked services.

Definition 4.3 (Component Composability) *Let $A = (\mathcal{C}, links, subs)$ be a component assembly, and C_i and C_j two components of A ($C_i \in \mathcal{C} \wedge C_j \in \mathcal{C}$). The components C_i and C_j are c-composable if all the links between sr_{C_j} and sp_{C_i} in A are defined between composable services:*

c-composable$(C_i, C_j) \Leftrightarrow$
$\forall (C_i, n_1, C_j, n_2) \in links \bullet$ *s-composable*$(\nu_i(n_1), \nu_j(n_2)) \vee$ *s-composable*$(\nu_i(n_1), \nu_j(n_2))$.

Note that the disjonction $\vee$ handles the direction of the *s-composable* relation (provided-required).

4.2.3 Assembly Composability

The composability of an assembly is a generalization of the composability of the pairs of components.

Definition 4.4 (Assembly Composability) *A component assembly $A = (\mathcal{C}, links, subs)$ be a component assembly is composable if each couple of components of A are c-composable:*
composable$(A) \Leftrightarrow \forall C_i \in \mathcal{C},\ C_j\mathcal{C} \bullet$*c-composable*$(C_i, C_j)$.

4.3 Interface Analysis: an overview

We define composability at different related levels (4.2): service level and component level. In our Kmelia component formalism, the interface of a component contains the sets of provided and required services (with the naming and typing informations); additionally, informations on required or called sub-services are attached to the interface. In a similar way, these informations are available for the service descriptions.

Accordingly, the static analysis of the interface of a component is achieved by using: *i)* simple correspondence checking algorithms and possibly standard typing algorithms; *ii)* deep investigation on the availability of required or called sub-services. The definitions given above are used to perform this static level analysis. At this stage, some incompatibilities may be detected. We cover by the way a main part of (static) interoperability properties and architectural properties.

In the case of properties 1.a, 1.c, the static analysis of the interface of a component may be achieved by using simple correspondance checking algorithms and eventually standard typing algorithms.

In the case of property 1.b, proofs on the assertion are necessary. One way to deal with the interface analysis is to translate our component into existing frameworks (FSM, Z, B, Process Algebra) and reuse the tools of these frameworks to tackle the analysis. This is now the standard way.

[3] Note that the service and parameter names can be different.

4.4 Behavioural Compatibility Analysis

In the following we focus on the *behavioural compatibility* aspect. The classical *safety* and *liveness* aspects apply obviously to software components. The safety is more concerned with the functionnal properties of the components. That means the correctness with respect to the needs. Temporal behaviour (liveness) is also an aspect related to correctness; a system should evolve and should perform its tasks in time.

But these properties should be adapted to component features, for example behavioural compatibility. The *behavioural compatibility amongst components* is a widely studied topic [35, 15, 9, 13]. behavioural introspection (discovering the component behaviour) is one way to deal with behavioural compatibility; but one has to prove compatibility. Checking behavioural compatibility often rely on checking the behaviour of a (component based) system through the construction of a finite state automata. However the state explosion limitation is a flaw of this approach.

The main concern is to check that a given component interacts correctly with another one (which may be provided by a third party developer). The interaction between components may involve not only two but many components. Assume that a verification of the *architectural properties* is already performed for a given component. This implies that each service of this component is completely described. Remind that each service is described with an eLTS where the transitions are labelled with guarded elementary actions and communication actions (see 2.1).

The component interacts correctly with its environment if its services are composable with the other services. But we consider only one caller service and one called service at time. We check that *Bp* a given eLTS matches with *Br* a second eLTS: *compatible(Bp, Br)*. A complete interaction between the services of several components results in a pairwise local analysis between the LTS of a caller and that of the called service. The eLTSs are unfold to obtain LTSs. Therefore, two services interact until a terminal state if the labels of their associated LTS are in correspondence according to a protocol that we have defined. The protocol that defines *compatible* is a set of rules based on the labels of the transitions going from a current state to the following states (output transitions). The rules indicate the correct evolutions according to the current states of two involved services: from a current state considered in each LTS, we explore the labels on the output transitions. In the case of elementary actions on the labels, each LTS evolves independently, their current states are updated. In the case of communication actions on the labels, the transitions match if for the considered services (hence the appropriate channels), we have the matching pairs: *send(!)-receive(?), call service(!!)-wait service start(??), emit service result(!!)-wait service result(??)*. In this case each LTS evolves in its next state. If the labels do not match, an incompatibility or a deadlock is detected.

After a final state of a called service, the caller may continue with independent transitions or with transitions that imply other (sub-)services. When the final states are reached without deadlock, the services are compatible.

4.5 A Verification Algorithm for behavioural Compatibility

4.5.1 Service Specification Analysis

In [5] we formally defined the compatibility between components by considering the interaction between the services of the involved components. We show on the studied exemple, how this analysis is achieved. Remind that the behavior (named $\mathcal{B}$) of a service is specified with an LTS $\langle S, L, \delta, \Phi, S_0, S_F \rangle$ (see section 2.1.2).

Therefore the evolution of a service is exactly that of its LTS. We assume some working hypotheses. When a service is called, its initial state becomes the current state (local to this service). The service is then an *active* service.

Several services may be simultaneously active; each one has its current state. When a service calls another one, both can interact by exchanging communication messages (emission or reception). Apart from the communications, the active services evolve one independently from the other.
From any state of the behavior of a system, one can carry out an internal action, a block of actions, a call to a service, a communication action, and a block of guarded actions.

Algorithm We built an algorithm to check the conformity of the interactions between services of the components. The algorithm is built starting from a set of rules. We defined the following rules for the evolution of the behaviors

of the services:
R_{ambig_a}**:** From a current state e_c, when there is more than one transition labelled with internal actions or with calls from service, there is an ambiguity. The analysis is not continued.
R_{act}**:** From a current state e_c where there is only one transition labelled by l representing an *internal action* α, the action α is carried out and the behavior of the service continues in the state $e_s = \delta(e_c, l)$.
R_{call}**:** From a current state e_c where there is only one transition labelled by l representing a *service call*, one checks that this service exists in one of the components of the environment, that this behavior has an initial state and a final state, then the behavior of the service continues in the state $e_s = \delta(e_c, l)$.
R_{comm}**:** From a current state e_c where there are one or more transitions labelled by a *communication action* of the $?c$ type or the $!c$ type, one checks that there exists an active service in the environment, which proposes the same communication actions: either $!c$ or $?c$; when it is the case, two symmetrical actions, one of each service, are carried out in the same time; the behavior of each implied service thus continues in the state $e_s = \delta(e_c, c)$. Here c indicates the communication action which is carried out.
R_{block_acts}**:** From a current state e_c where there is only one transition labelled by a *block of actions* ba, one observes inductively the previous rules; the behavior of the service continues in the state $e_s = \delta^*(e_c, ba)$. Here, δ^* indicates a successive application of δ (compared to the actions contained in ba).
R_{guard_acts}**:** From a current state e_c where there are one or more transitions labelled by a *block of guarded actions* g_acts, one applies inductively the preceeding rules to the block of actions located after the guard, and for each transition. It results therefore to the rule R_{block_acts}.

The skeleton of the algorithm is as follows:

ALGORITHM VerifInteraction(B) = /* B is a transition system */
BEGIN
ec := *CurrentState*(B)
DO
Apply one of the rules in
{ R_{act}, R_{call}, R_{comm}, R_{block_acts}, R_{guard_acts}}
UNTIL
no rule is applicable OR
an error occurred
DONE
END

4.5.2 Interaction Analysis within our Example

We apply the algorithm starting from the call to the service `Retrait(carte)` from the component `BASEGAB`.

1. The service is called by `Retrait()` from the component `IHM_CLIENTR`. The call of the service `Retrait(maCarte)` is performed from the initial state `e0` of `IHM_CLIENTR`.
2. The rule R_{appel} is applied, the transition system assocated to `Retrait(carte : Carte)` exists, this service becomes active and its current state is `i`.
3. The state of the service `Retrait()` is now `e1`. Note that from this state, the labelling `<Code> <Montant>` indicates that one can eventually invocates the services `<Code>` or `<Montant>` from the `IHM_CLIENTR` component.
4. Let us follow the evolution with the `Retrait(carte : Carte)` service.
5. From the `i` state we reach the `e0` state with the R_{act} rule.
6. From the `e0` state we reach the `e1` state with the R_{appel} rule, therefore `Code()` is activated and its initial state is considered.

7. From `e1`, we reach the `e2` state using the R_{block_acts} rule; we proceed with an intermediary state which involve a simultaneous transition (the R_{comm}) rule from `Code`, this one terminates; the control is now in the state `e1` of `Retrait()`.
8. From `e2`, there are three guarded state; therefore we apply the R_{gard_acts} rule to reach the `e0` state or the `e3` state, or the `e4` one. The guarded state are call to servicess; these services are available in the environement, we do not go to more details.
9. From the `e3` state, R_{gard_acts} is applied again, we reach either the `e4` state, or the `e5` one according to the considered transition.
10. From `e4` we reach the `f` state with the rule R_{comm}, simultaneously with a transition from `e1` to `e2` in `Retrait`. The latter is terminated.
11. From the `e5` state, we reach the `e6` state using R_{appel}. The `Montant` service is then activated; it interacts with `Retrait`.
12. From `e6`, we reach `e7` using R_{comm}, with a communication action (`resmontant`) performed simultaneouslyt with the `Montant` service.
13. From the `e7` state, we apply R_{gard_acts} to reach the `e3` state.
14. From the state `e7`, we apply R_{gard_acts} to reach `e8`.
15. Finally from `e8`, we reach the final state `f` using R_{comm}, simulatneously with an action performed in `Retrait`.

In short, for our example, the interactions are in conformity; there was no error. If not, one could thus have corrected them.

We generalized this algorithm in order to check independently the conformity of the interactions of the environment of the components.
The principle (see [5] for the details) is as follows: for a given service, we defined in a general way the behavior which is necessary for a correct interaction; any behavior similar to this necessary behavior is thus in conformity.

The necessary behavior is defined according to the labels of the transitions; for example, a reception action is necessary for an emission action and vice versa; a service call requires a (sub)system of transition associated with this service, etc.

This approach makes it possible to be ensured of the compatibility of the interactions in the assemblies of components and to avoid also the problem of the state explosion.

4.6 Implementation

A first implementation [33] of the algorithm is achieved using Java. In this implementation, we develop a prototype which considers only independant services. Each service is given within a separated text file.
The prototype is made of three modules:

- a syntatic analyser based on antlr,
- a graph structure manager and
- the compatibility verifier.

Some tests are performed with this prototype. They are quite satisfactory according to the considered version of the Kmelia formalism.

This experimentation demonstrates the feasability of the analyis with correct complexity (it is polynomial on the graph size) of the Java program.

However, this first implementation is done according to a preliminary Kmelia formalism. The latter is now considerably improved, therefore the prototype should be upgraded.

5 Translation of Services into Lotos

5.1 Introduction to Lotos

Lotos [21] is an ISO standard formal specification language. It is initially designed for the specification of networks interconnection (OSI) but is also suitable for concurrent and distributed systems. Lotos extends the process algebra CCS and CSP and integrates (algebraic) abstract datatypes. Hence Lotos is a process algebra; a Lotos specification is structured with process behaviours. It has the main behaviour description operators of the basic process algebra CCS and CSP. Lotos uses the "!" and "?" operators of CSP which denote respectively emission and reception.

The salient features of Lotos are: the powerful multi-way synchronisation; the use of communication channels called *gates*; the synchronous interaction of processes; the use of algebraic data types to model data part of systems; the availability of a toolbox (CADP [16]).

A process is the description in the time, of the observational behaviour of a given system. The description is given as the non-deterministic combination of the sequence of events feasible by the system. The set of events of a behaviour is called the *alphabet* of the process.

In a process specification, a sequence of events is denoted with ";"; the choice between alternative behaviours B and C is described with "`B [] C`"; `[Bterm] -> B` describes a process behaviour B guarded with a boolean term `Bterm`; the inaction is denoted with `stop`; a successful termination is denoted with `exit`; the sequential composition of behaviours `B` and `C` is described with `B >> C`.

Three parallel composition operators are used to compose processes: `|||` is used for the interleaving behaviour of the composed processes; `||` is used for the strict (on all the events) synchronisation of the involved processes; `|[L]|` where `L` is a synchronisation list (of events) is used to synchronise the processes on the events within the list L; when L is empty this results on a interleaving.

Both synchronous and asynchronous communications may be described in Lotos.

A Lotos *event* stands for a synchronisation between two or more processes. An event is *atomic*. There are three kinds of synchronisation: the *pure synchronisation* where no value are exchanged between the involved processes; the *value establishment* where one or more processes supply a value passed to other processes; the *value negotiation* where one or more processes agree with a set of value.

The ISO Lotos has an operational semantics in terms of labelled transitions systems. The semantic rules define the behaviour of the Lotos processes and their communication. For the data part, algebraic term rewriting is considered to evaluate data terms and each variable may be instantiated by the values corresponding to its type.

5.2 Translating the Service Automata into Lotos Processes

An *output transition* of a given state is a transition going from this state to another one. An *input transition* is a transition coming from any one state and entering another considered state.

Remind that each service of a component is described with a transition system. The transitions are labelled with: service calls, elementary actions, guarded actions, communication actions. Each state has an identifier. Some of the states are additionally labelled with a list of action names (those actions which can be called when the current evolution reaches the concerned state).
In order to manage flexibility of interaction and also to tackle service complexity, sub-services are accessible by means of branching from some nodes which are annotated with the name of sub-services.

The general principle of the translation is that the transition system which describes a service is expressed with one or several Lotos processes: one main process is associated to the service and one or several subprocesses are associated to the former one.

Basically, each state is translated into a process. From each state of the service there are one or several transitions going to other states. This is translated by a choice between as many process behaviour as possible in the Lotos process.

The behaviour of a component service is expressed as a combination of actions which can be: internal actions, service calls or communications. These actions label the transition from one state to another one. Therefore from each state we translate the related transitions.

General Translation Principle

We give here the main principles which are detailed in the subsequent subsections. The considered hypotheses for the translation of service automata are the following.

i) To deal with the communication, each service has a *default channel* made by prefixing the service name with the word "chan_". Thus a service which is named *serv* for example, has a channel named `chan_serv`. This channel is used as a parameter of the process corresponding to the service. In the same way, the channels associated to the services with which a service $serv$ communicates (service calls appearing in the behavior) are listed as parameters.

ii) We treat the activation of a service with a communication (to enter the initial state of the called service). A process corresponding to a service waits for a call. The caller service sends a call.

Initially each service (the associated process) waits for a communication using its default channel.

A caller service calls a service by sending a message (with the called name as parameter) on the default channel of the called service. The parameters are also sent using the default channel of the called process.

As a process describes a state machine, the translation from a state machine to a process is quite straightforward. Each state is described by a process. The behaviour of the latter describes the transitions which are attached to the corresponding state.

In our case (with Lotos processes) the translation procedure is performed as follows: each service state gives a Lotos process; the translation is then achieved by considering each state of the service. Each output transition of the service corresponds to an action in the Lotos process. It may be followed by the translation of the reached state.

A state with several output transitions is translated with a nondeterministic choice of the translation of each output transition.

A state with more than one input transition is translated with a subprocess. Indeed, having more than one input transitions means that the state can be reached from several transitions, therefore the subprocess is reused from different state translations.

A state annotated with a list of service names is translated by a nondeterministic choice between several subprocesses. Each subprocess corresponds to the interaction with one of the listed services.

Service calls are treated by means of a communication: an emission statement. We distinguish the communication operators used for service from the classical ones; therefore !! and ?? are respectively used for an emission statement and a reception statement. When calling a service, the name is passed as a parameter; thus a call to a service without a parameter results in sending the action name on the channel: called_chan!!called_name. Likewise the result of a service is also passed by means of a communication. The caller service waits on the default channel; the called service sends the appropriate value(s) on the same channel.

An elementary action is translated with a symbolic (or an internal) action. As far as the *guarded actions* are concerned, first the guard are abstracted as an atomic element and then the guarded action gives a sequence of actions.

Communication actions are translated with Lotos communication actions.

To enable a new call of a terminating service from one or several services, we complement the behaviour of the process by a looping transition labelled with the internal action i which reaches the initial state. Thus the process can be called again.

According to the preceding statements, rather than a straightforward translation, we have a specific semantic encoding (namely $LotosEncoding$) of the service specifications. Briefly, the encoding into Lotos of service specifications is inductively performed by considering:

- service interface without formal parameters,
- service interface with formal parameters,
- service states (initial, final, intermediary and annotated) and
- service transitions.

The encoding is mainly achieved by considering first the state of the service specifications and then the treatment of transitions.

The data used within the description of services are also considered within the encoding into Lotos.

5.3 Data Translation

The following policies are considered to manage data during the translation into Lotos.

- We use enumerated or byte types to express data; accordingly the data are reduced, but we avoid the state explosion problem.
- For each service $ServName$, we define a Lotos datatype named `MsgTypeServName`. It has a constructor which named with respect to the service; this permits the call of the service. Besides, all the messages which are sent to the default channel associated to a service are used as constructors of the data type associated to this service.
- Enumerated data are translated with constructors of abstract data types.
- The expressions used within action are translated as a simple actions in the Lotos process. The expressions are not evaluated.
- Each guard is encoded as a simple action; therefore each guarded transition has a corresponding behaviour (sequence of actions) in the Lotos process. The guards are not evaluated.

5.4 Encoding of service into Lotos

We define a set of semantic encoding rules to support the translation into Lotos of the component services. These semantic rules permit a systematic translation.
Three kinds of encoding rules are defined: service interface translation, state translation rules (denoted by the LotosEncoding procedure) and transition labels translation rules (denoted by the LotosEncodingL procedure).
We do not give a full description of these rules, but we give some of them to illustrate the used approach. All the rules are formalised using Structural Operational Semantic (SOS) rules.

Encoding Service Interfaces (rule SI)

A service without formal parameters is called by sending its name in the channel attached to the service. Hence a service `servName()` is encoded by

```
process servName[servName_chan, ...]: exit :=
   servName_chan? snx: MsgTypeservName;[snx = servName];
```

A service with some formal parameters is encoded by a process which waits for the name of the service that is encoded and all the parameters on the channel dedicated to the service (its default channel and the channels of the services with which it communicates). Hence a service `servName(p1: T1, p2: T2, ...)` is encoded with:

```
process servName[servName_chan, ...]: exit :=
   servName_chan? snx: MsgTypeservName;
       ? p1: MsgTypeservName;
          ?p2: msgTypeservName; [snx = servName] ->
            ...
```

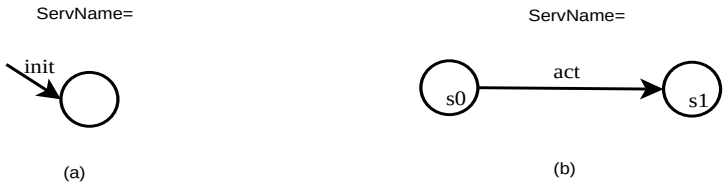

Figure 1: Initial and simple transitions

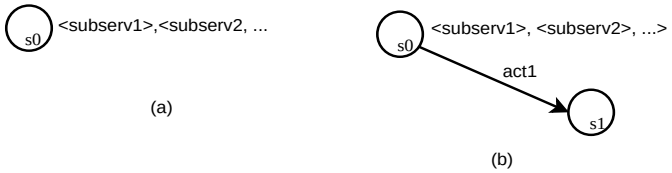

Figure 2: Branching Node

Encoding Service States (rule SS)

The encoding into Lotos of the state described in the Figure 1.(a) is as follows. This corresponds to the initial state of the process of a service.

```
Process ServName[ServName_chan, ...]: exit :=
   ServName_chan? msg: msgTypeServName [msg = servName] ; exit
Endproc
```

Encoding Terminal States (rule TS)

The encoding of a terminal state is:

```
i; exit
```

To enable the recalling of a service after a terminating call, the behaviour returns into the initial state with the >> operator.

```
( i; exit) >>  servName[ServName_chan, ...]
```

Note that this encodes a non-exiting Lotos process. A variant of this encoding enables us to have an terminating process. This one is used to perform analysis of the generated code: it either terminates or loops.

```
( i; exit [] servName[ServName_chan, ...])
```

Encoding Branching States (rule BS)

The encoding of 'branching node' (see Fig. 2) is achieved according to the default channel of the services which appear on the annotated nodes. Consider that a given state ss is annotated with `<subserv1>` and `<subserv2>`. If the service `subserv1` is called, then the current service proceeds within the `subserv1` at its initial state. In the following let $initState(serv)$ denotes the initial state of a service $serv$.

The encoding into Lotos of a branching node (see Fig. 2.(a)) is as follows:
LotosEncoding(`s0`) =

```
Process SP_Process_s0[...]: exit =
   (  chan_subserv1?fprm: MsgTypesubserv1 [fprm = subserv1];
         LotosEncoding(initState(subserv1))
   [] chan_subserv2?fprm: MsgTypesubserv2 [fprm = subserv2];
```

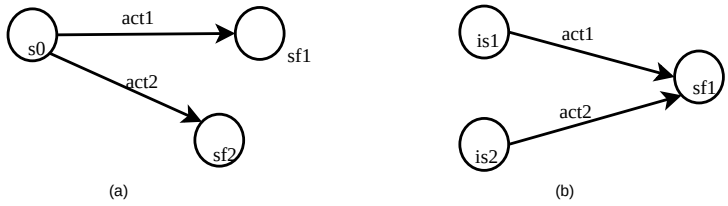

Figure 3: State sharing transitions

```
            LotosEncoding(initState(subserv2))
    )
Endproc
```

The encoding into Lotos of the behaviour of the Figure 2.(b) is : LotosEncoding(`s0`) is

```
Process SP_Process_ss[...]: exit =
    (  chan_subserv1?fprm: MsgTypesubserv1 [fprm = subserv1];
            LotosEncoding(initState(subserv1))
    [] chan_subserv2?fprm: MsgTypesubserv2 [fprm = subserv2];
            LotosEncoding(initState(subserv2))
    [] act1; LotosEncoding(s1)
    )
Endproc
```

Transition Labels Translation Rules : *LotosEncodingL*

An *elementary action* (act) is translated with an abstract action (aa). Informally LotosEncodingL(`act`) = `aa` where `aa` is an abstraction (i.e. an element of the process alphabet) of the action `act`.

As far as the *guarded actions* are concerned, first the guard is abstracted as an abstract action and then the action behind the guard is encoded; this gives a sequence of actions. The formal SOS rule is as follows:

$$\frac{\begin{array}{c}[guard]\ act \ \in \ L \ \wedge \\ \texttt{absguard} = \text{LotosEncodingL}(guard) \ \wedge \\ \texttt{aa} = \text{LotosEncodingL}(act)\end{array}}{\text{LotosEncodingL}([guard]\ act) = \texttt{absguard; aa}}\ \textbf{rule GA}$$

A service call (using `!!`) is treated by means of a communication: an emission statement (using `!`). Likewise the result of a service (using `??`) is treated by means of a communication (using `?`). The caller service waits on the default channel; the called service sends the appropriate value(s) on the same channel.

Communication actions are simply translated with Lotos communication actions using appropriate channels.

Encoding Service Transitions (rule ST)

The encoding into Lotos of the transition in the Figure 1.(b) is achieved as follows:

```
LotosEncodingL(act); LotosEncoding(s1)
```

Note that if `s1` is a terminal state we will have the encoding `act; exit` for the service depicted in Figure 1.(b). To symplify, we may consider `LotosEncodingL(act)=act`.

The encoding into Lotos of the transitions in the Figure 3.(a) corresponds to the encoding of `s0`. The result is as follows:
LotosEncoding(`s0`) =

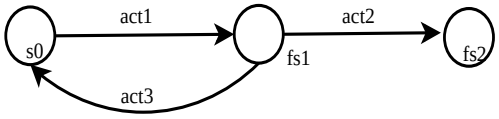

Figure 4: Looping transitions

```
(
   LotosEncodingL(act1); LotosEncoding(fs1)
 []
   LotosEncodingL(act2); LotosEncoding(fs2)
)
```

The encoding into Lotos of the transitions in the Figure 3.(b) corresponds to the encoding of `is1` and `is2`. This results in:

LotosEncoding(`is1`) =

```
Process SP_Process_is1[...]: exit =
  LotosEncodingL(act1); LotosEncoding(sf1)
Endproc
```

and
LotosEncoding(`is2`) =

```
Process SP_Process_is2[...]: exit =
  LotosEncodingL(act2); LotosEncoding(sf1)
Endproc
```

Encoding Looping Transitions (rule LT)

The encoding of a looping transition (see Fig. 4) is like the other transitions except that the final state of a given transition is already treated. According to the schema described in the Figure 4, we have the following encoding:

LotosEncoding(`s0`) =

```
Process SP_Process_s0[...]: exit =
  LotosEncodingL(act1); LotosEncoding(fs1)
Endproc
```

and
LotosEncoding(`fs1`) =

```
Process SP_Process_fs1[...]: exit =
(
 LotosEncodingL(act2); LotosEncoding(fs2)
[]
 LotosEncodingL(act3); SP_Process_s0[...]
)
Endproc
```

Encoding Final Transition (rule FT)

This encoding results from the application of previous encoding rules: transition encoding and final state encoding. Thus the encoding of the transition of the Figure 5 results from the combination of LotosEncoding(`s0`) and LotosEncoding(`s1`). This gives:

```
act; exit
```

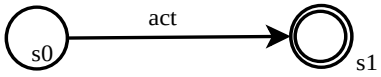

Figure 5: Final transition

5.5 Formalisation

All the previous encoding rules should be formalised. We give here the main lines of the formalisation process.

Consider in the following the services of a component. For a service $serv$, let δ denotes the function which gives the behaviour of a service $serv$; $states$ denotes the function which gives the states of a service $serv$; $initial\_states$ denotes the function which gives the set of initial states; $final\_states$ denotes the function which gives the final states.
The encoding of a state is of a transition of $serv$ is formalised with:

$$\frac{(is, act, fs) \in \delta(serv) \quad \wedge \quad \texttt{SP\_Process\_fs} = \mathsf{LotosEncoding}(fs)}{\mathsf{LotosEncoding}(is) = \texttt{act; SP\_Process\_fs}} \textbf{rule ST}$$

The formalisation of the previous terminal state encoding rule is the following.

$$\frac{fs \in states(serv) \quad \wedge \quad fs \in final\_states(sf)}{\mathsf{LotosEncoding}(sf) = \texttt{i; exit}} \textbf{rule TS}$$

The other rules are formalised in the same way.

5.6 Encoding Labels of Transitions

We examine here the encoding of the actions which label the transitions of the service behaviour.

rule LabA: An elementary action (assignment, ...) is encoded by a simple Lotos action. The name of the latter is generated from a dummy string used for the alphabet of the current service.

rule LabG: A guard `[boolterm]` is also encoded by a simple action. It is not evaluated. In the same way as for elementary action, the name of the action is generated from a dummy string used for the alphabet of the current service.

rule LabE: An emission action `chanName!term` is encoded in the the same way in the corresponding Lotos process.

rule LabR: A reception action `chanName?term` is encoded in the the same way in the corresponding Lotos process.

rule SC: A service call `ServName!!term` is encoded with an emission action (that is a communication) on the default channel of the service, except that the channel name is modified according to the default channel name of the service: `ServNname_chan!term`

rule SW: A waiting for a service call `ServName??term` is encoded with a reception action (that is a communication) on the default channel of the service, except that the channel name is modified according to the default channel name of the service: `ServNname_chan?term`

5.7 Examples of Complete Encoding into LOTOS

The service $Code()$ depicted in the figure 6 is translated as follows:

Figure 6: Example of Services

```
  PROCESS Code [Chan_Code,Chan_withdr] : exit :=
     Chan_Code?xmsg: MsgTypeCode [xmsg=Code] ;
         SP1_Code [Chan_Code,Chan_withdr]
  ENDPROC

  PROCESS SP1_Code [Chan_Code,Chan_withdr] : exit :=
    Chan_Code!resCode ! idCod1 ;
        ( i ; exit [] Code[Chan_Code, Chan_withdr])
  ENDPROC
```

The service $AskMoney()$ depicted in the Figure 6 is translated as follows:

```
PROCESS AskMoney [Chan_DR, Chan_withdr, Chan_Code, Chan_Amount]: exit:=
    Chan_DR ? xmsg : MsgTypeAskMoney [xmsg = AskMoney] ;
     (*e0*) SP1_AskMoney [Chan_withdr, Chan_Code, Chan_Amount]
 WHERE
 PROCESS SP1_AskMoney[Chan_withdr, Chan_Code, Chan_Amount] : exit :=
 (*e0*)
   Chan_withdr!withdrawal !idcard1 ; (*e1*)
     SP2_AskMoney[ Chan_withdr,  Chan_Code, Chan_Amount ]
 ENDPROC

 PROCESS SP2_AskMoney [Chan_withdr, Chan_Code, Chan_Amount]: exit :=
  (*e1*)
  (Chan_withdr ?xmsg: MsgTypeRetrait ? b: Bool [xmsg = result]; exit
   [] Chan_Code?xmsg: MsgTypeCode [xmsg=Code] ;
               SP1_Code [Chan_Code, Chan_withdr]
   [] Chan_Amount?xmsg: MsgTypeAmount [xmsg = Amount] ; i ;
               SP1_Amount[Chan_Amount, Chan_withdr]
   )>> (i; exit [] SP2_AskMoney[Chan_withdr, Chan_Code, Chan_Amount])
   (*e2*)
  ENDPROC
ENDPROC (* of AskMoney *)
```

5.8 Using Lotos for the Compatibility Verification

The behavioural compatibility checking is based on LOTOS processes communication. We use the `|[L]|` composition operator. The compatibility verification turns in checking that the processes that represent the services communicate perfectly. As far as simple actions are concerned, each process evolves independently from the other. But communication actions should be coordinated. An emission action should correspond to a reception action and vice versa.

A pair of services is involved in a compatibility check, the caller and the called one; for example `behaviour` and `withdrawal` in our case (see Fig. 2). Note that the `withdrawal` is required by `behaviour` via the name `ask_for_money`. A renaming of `withdrawal` with `ask_for_money` is performed.
These two services (the caller and the called) are translated into LOTOS processes (say `Lbehaviour` and `Lask_for_money`);

each process has its alphabet (`alphabet` in the phollowing); the processes are then composed using the `|[L]|` operator to get a resulting process called `Res` in the following. `L` is instantiated with the list of channels used for the communication between both services as illustrated above.

```
Res =  Lbehaviour[alphabet]
    |[chan_behaviour, chan_ask_code, chan_ask_amount]|
       Lask_for_money [alphabet]
```

Consequently, the services are compatible if the obtained `Res` process has no conflict according to the composition operator.
As far as the running example is concerned, we check that `USER_INTERFACE` and `ATM_CORE` are composable according to the services (`ask_for_money, withdrawal`): the interface checking is easy. The behaviours of `ask_for_money` and `withdrawal` are compatible.

5.9 Implementation

To make it easy the experimentation of our component model, we develop a prototype (named `kml2lotos`) to translate the component services into LOTOS is also developed using Java. The `kml2lotos` module is a part of the general toolbox under development to support our model. The current translator uses the output of an analyser module.
Given an input component specification (in kmelia), the analyser parses the specification and generates the corresponding internal structure. The latter is read by the `kml2lotos` prototype; it generates communicating LOTOS processes which are used as input to the CADP toolbox. As far as the previous ATM case study is concerned (see Section 2), the experiment deals with an assembly of components. Specific services (a caller with a called one, branching node with the sub-services) are checked. The CADP raises failures when there are lack of channels, wrong channels, incompatible types, blocking or incompatible behaviours. The experiment using CADP helps us to discover specification errors; for example when a wrong communication channel is used. When the errors are recovered and the communications are fine, the CADP `caesar` utility generates the (execution) graph corresponding to the system. The graph is very large in the case of brute translation; but when we erase independent alphabet actions and minimise the generated graph, we get a graph with less than hundred states. Stepwise simulation (using CADP `executor` utility) is performed to analyse the evolution of the system.

6 Translation of Services into MEC

This section reports experimentations on verification of behavioural compatibility using the MEC verification tool for state transition systems. The behavioural compatibility is the property of correct interaction between two services in a service context (see Section 4.2). The verification is twofold: translation into MEC, verification of the properties by the MEC model-checker.

This section is organized in two parts: short overview of MEC, basic transformations and extensions. A subsection is dedicated to each part, respectively sections 6.3 and 6.4. In each case, we study the translation into MEC and the dynamic properties proofs. Section 6.3.4 is to prove inconsistencies in the case study.

6.1 Aims and Scope

We do not check an overall component model. The goal is to check the behavioural compatibility of each provided service

1. against all the required services that call it: peer to peer component behavioural compatibility (this is a local property),

2. against all the services that it requires: the service is fully and correctly specified (this is a global property).

The same translation can be used in both case. What varies is the workspace.

In the following we focus on the first verification only. The behavioural compatibility of each provided service (the reference service) is checked in the context of one of its caller (component requiring this service through an assembly). The (verification) service context is made of a reference service, a calling service, a required service that links them, and the components that include them. In the ATM case study, the reference service is `withdrawal`, the calling service is `behaviour`, the link is (`withdrawal`, `ask_money`), and the context includes the `USER_INTERFACE` and `ATM_CORE` components This context is given in figure 4.

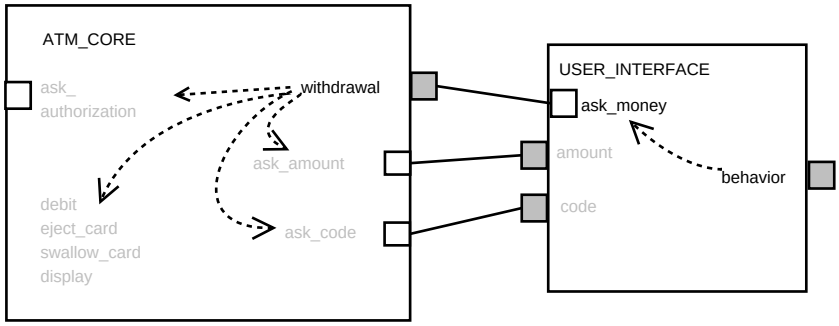

Figure 4 : *A behavioural Verification for an ATM System*

In short, the behavioural peer to peer compatibility verification processes as follows:

```
for each link (ps, rs) do
  /* ps = provide service and rs = required service */
begin
   p := component(ps) ;
         /* the component p holds the provided service ps */
   q := component(rs) ;
         /* the component q holds the required service rs */
   if (p <> q) then
         /* only external links are explored for binary compatibility */
   begin
      for each cs in calling(rs) do
         /* the behaviour depends on the calling context */
      begin
           check_behaviour(ps, rs, cs, p, q)
           /* checks peer to peer behaviour compatibility */
      end;
   end;
end;
```

The rest of the section is dedicated to the implementation of the $check\_behaviour$ procedure in MEC.

6.2 MEC

MEC [7] is a model checker for labeled transition systems (LTS) (or state machines). MEC LTS are either simple LTS or synchronization systems of LTS.

- A simple LTS is defined by a list of transitions, the union of the source states and target states of the transitions defines the set of states. States and transitions are annoted by boolean parameters or properties called marks. Each mark hence defines a subset of transitions or a subset of states. For example, `initial` defines the set of initial states.
- A synchronization system is a synchronized product of transition systems[4]. The synchronized product is a cartesian product restricted to the transition labels belonging to the synchronization constraints (synchronization vector). MEC computes the synchronization products.

[4] A synchronized product of transition systems and synchronization systems would be more powerful and helpful.

Model checking with MEC is to compute properties (marks) on transitions or synchronization systems. Each property is defined as a set of states or transitions, those annoted by the mark of this property. MEC includes powerful graphs algorithms, including fixpoints computation and user-defined functions.

We currently work with the MEC 4 model checker. A new release, called MEC 5 is available with the Altarica tool [8]. It is discussed in section 6.4.5.

6.3 Basic Transformations

This section describes the transformation of service behaviours into MEC transition systems and service contexts into synchronization systems. The translation algorithm is detailed in section 6.3.2, then the specification properties are explored in section 6.3.3. The proposal is illustrated on the ATM example.

6.3.1 Workspace Computation

The workspace is the set of services which are concerned by the verification of a service. For a global service verification it includes all the services called by the reference service, the service context and the related sub-services.. For a peer to peer verification the workspace includes the reference service, the calling service, the link between them and all the secondary services (belong to the component of the calling service and are called by the reference service). This context is shown in figure 4. In any case, the semantic interpretation is communicating processes, each service execution is a process. Each service of the workspace maps to a process and is translated into a MEC transition system.

Each service call that do not belong to the workspace is considered to be simple action and treated as such. Thus the search space is limited to a finite number of processes. In our example, the required service `ask_authorization` is supposed to be correct[5].

6.3.2 Translation

The formalism of service behaviours is more expressive than the one of MEC, thus the translation is not so immediate. Some service behaviour concepts can translated by the combination of several MEC concepts (channels, branching states and transitions). Others cannot be translated easily (guards, parameters). The translation is based on an algorithmic approach and (customizable) translation rules. The current algorithm is applied manually, an implementation will be coded once the grammar of the source model will be operational. The translation is presented by difficulty level and by step.

Translation of the Model Kernel

Each service of the workspace is interpreted as a process and translated by a transition system. The service parameters (definition and call) are omitted because there a no simple way to translate them into MEC. The model kernel also ignores the guards and the communication parameters and results. We assume that each service is instanciated once and each provided service ends by a final state. Handling several instances of a service is studied in section 6.4.3.

Syntactic conventions are held during the translation because MEC does not accept some characters (spaces, quotes, parenthesis, punctuation marks, numbers, signs...). For example, the dotted notation is replaced by a '_' notation. We can also limit the length of labels to 15 characters.

1. We assume that the services are identified by their names and not their signature (name + parameters). Each service behaviour is translated by a MEC transition system whom name is composed from the component name and the service name. The service `ATM_CORE. withdrawal(card:CashCard)` is translated by the LTS `ATM_CORE_withdrawal`.

    - An initialization transition, labeled by `start_` concatenates with the service name, is added to synchronize the service call. The new initial state is named `init`.

[5] In fact, it will be checked later, when verifying the `authorization` service

- The transitions that target to the final state are treated as ordinary transitions.
- For each state, an empty transition `e` (that represents the empty action ϵ) is explicitely added. It is used in the synchronization constraint to make the LTSs evolve independentdly when no communication occurs.

2. Each service state is translated into a LTS state (naming rules applied for special characters).
   - The initial states are marked to be initial (at least one for MEC).
   - The final states are marked to be final in MEC. It is used to keep them away from deadlock states.
3. Each transition is translated into a LTS transition:
   - The initialization transition is labeled by `start_` concatenates with the service name.
   - The transitions corresponding to internal actions are labeled
     (a) sequentially (t1, t2...) using a map table between labels and full original names. In this solution, a LTS is hardly readable.
     (b) by renaming labels according to the MEC naming rules. The problem is to handle conflicting renamings, using a table of identifiers.

     The second solution has been choosen for our experimentation.
   - Service calls are transitions labeled by the required service name. The link between a required service and a provided service is solved in the assembly translation, see section 6.3.2.
   - A communication includes a service name (gate), a direction (? ou !) and a message. The following translation convention are adopted: a communication `serv!msg` is translated to an `emit_serv_msg`, a communication `serv?msg` is translated to an `rcv_serv_msg`. The default gate name is the reference service. Recall that parameters are omitted. The correspondence between emissions and receptions (notion of synchronization and communication in the component model) is explained in section 6.3.2.

In our example, the state `e2` of service `behaviour` of the component `USER_INTERFACE` is considered to be final. This makes the termination properties more easy to compute.

Flat LTS
The service behaviour allows branching states and transitions. Since MEC does not support hierarchical LTS the state machine has to be flattened. Three cases occurs:

- the branching (or compound) transitions (regular expressions of labels) are extended and generate new intermediate states
  - a sequential operator leads to a sequence of transitions, each intermediate state is labeled by the source state suffixed by an arbitrary sequential number. For example, a transition `e20 - a ; b ; c -> e21` is translated into 3 flat transitions `e20 - a -> e20_1`, `e20_1 - b -> e20_2` and `e20_2 - c -> e21`.
  - a colateral operator leads to a parallel composition of sequential transitions (in any order. For example, a transition `e20 - a , b , c -> e21` is translated into 6 sequential transitions `e20 - a ; b ; c -> e21`, `e20 - a , c , b -> e21`, `e20 - b , a , c -> e21`... Each sequential transition is translated along the previous rule. For sake of simplicity, each colateral operator can be simplified by a sequential one.

  Note that the intermediate states have no empty transitions since we assume that the branching transition is atomic.

- The branching states (nested (sub)services attached to the states) are a shortcut for ring transitions whom initial source state and final state are the related state. The branching states are expanded according to the following rules:
  - The initial source state and final state are the related state.
  - The new intermediate states are labeled by the source state suffixed by an arbitrary sequential number (see the coumpound sequential transition above).

  In our case study, the service `behaviour` of the component `USER_INTERFACE` is expanded on the state $e1$ to include the state machines of the subservices `code`.

- The synchronous service calls (:=) are translated by a sequential branching transition: call, result reception, assignment (internal action). It is treated as a sequential branching transition.

Synchronization Constraint (vector)
The synchronization vector indicates what actions should be synchronous. For more information on *communicating state machines* see [6, 7] or the chapter 5 of [3]. A similar concept exists for communicating process (see the LOTOS experiments in section 5). In MEC, the synchronizations apply to labels and not transitions themselves. During our experiments, we consider two situations: sequential system, parallel system. The differences are shown below.

Sequential System
Only one action occurs at each time in the system. This action may concern several LTS and thus synchronizes their 'local' actions. If a LTS is not concerned, an empty transition (the predefined 'e' label is a naming convention for the usual ε-transitions in non-deterministic LTS) occurs on it. The following rules hold:

- Each internal action `i` of a service behaviour `A` leads to to a (synchronization) line in the synchronization vector (also called the synchronization constraint) such that the cell for `A` is labeled by `i` and the other cells are labeled by the empty action `e`.
- Each service call `call_S` of a service behaviour `A` is synchronized with a service initialization `start_S` of a service behaviour `S`, the other cells are labeled by the empty action `e`. The initialization of the calling service (`behaviour`) in our example, is assumed to be an internal action to begin the verification.
- Each message reception `rcv_S_msg` of a service behaviour `A` is synchronized with a message emission `emit_S'_msg` of a service behaviour `A'` and the other cells are labeled by the empty action `e`.

Note that the correspondence between the names `call_S` of a required service and `start_S` of a provided service `S` is solved by the links of the assembly. The MEC specification of the sequential approach is given in appendix A.1.1.

Parallel System
In the parallel version, several actions may occur simultaneously, since each service executes only one action (no conflict in the synchronization vector). It means that the services are sequential but the global system is parallel. We can compute automatically the synchronization vector of the parallel version by a recursive combination algorithm :

```
SSV := sequential synchronization vector
PSV[1] := SSV /* synchronization vector */
i := 2 /* iteration variable : number of actions in parallel */
repeat
   PSV[i] := union(PSV[i-1], combine(PSV[i-1],SSV))
   /* combine tries to sum each line PSV[i] with each line of SSV
       the sum is valid if there is no conflict,
```

```
        i.e. empty action combines with any action */
until PSV[i] = PSV[i-1] ; /* fixpoint */
```

We illustrate the idea by the following example. Let a synchronization system SSseq composed from two LTS TS_A and TS_B, synchronized by a sequential synchronization constraint.

```
transition_system TS_A < width = 0 >;
init |- e -> init,
     action_a -> e0;
e0 |- e -> e0,
     action_a -> e0,
     stop_a -> e1;
< initial = { init } ; final = {e1} >.

transition_system TS_B < width = 0 >;
init |- e -> init,
     action_b -> e0;
e0 |- e -> e0,
     action_b -> e0,
     stop_b -> e1;
< initial = { init } ; final = {e1} >.

synchronization_system SSseq < width = 2 ;
    list = ( TS_A , TS_B ) >;
( action_a . action_b );
\* action_a and action_b are synchronous *\
( stop_a . e );
\* start_a is asynchronous *\
( e . stop_b ).
\* fin_b is asynchronous *\
```

To get the parallel version, we combine the lines of the sequential synchronization vector: line 1 cannot be combined with line 2 because there is a conflict on TS_A, line 1 cannot be combined with line 3 because there is a conflict on TS_B, line 2 can be combined with line 3 because there is no conflict in the synchronization vector. There are no other combination. The remaining parallel synchronization system is:

```
synchronization_system SSpar < width = 2 ;
    list = ( TS_A , TS_B ) >;
\* one action only *\
( action_a . action_b );
\* action_a and action_b are synchronous *\
( stop_a . e );
\* start_a is asynchronous *\
( e . stop_b );
\* fin_b is asynchronous *\
\* two actions *\
( stop_a . stop_b ).
\* combined line *\
```

The resulting state machine is shown in shown in figure 5. The gray part represents what is added in the sequential machine (black) to build the parallel version.

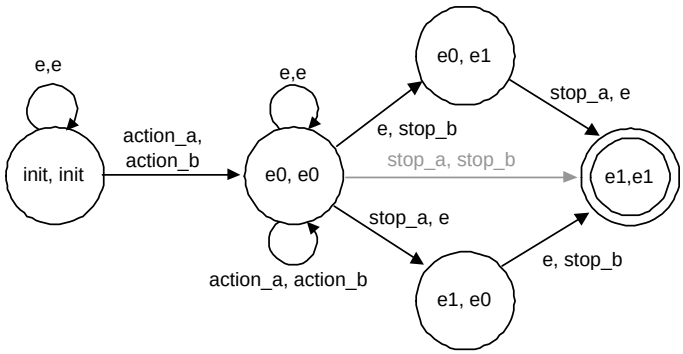

Figure 5 : *A Parallel Synchronization*

Note that the paralleling always adds only transitions but no states, whatever transition system we have. The synchronization is completed by new lines, grouped by number of parallel actions: one action, two actions, three actions...

In the ATM case study, only internal actions can be merged, the other combinations lead to a conflict. The resulting system is presented in appendix A.2.1.

6.3.3 Verification of Dynamic Properties

MEC is a model checker, it means that MEC computes every situation (every reachable state for the system) and computes properties on these situations. For short, MEC computations are based on graph algorithms and set operations that mark the states and transitions by properties. The result of a MEC computation is a set of state or transitions, which is interpreted in a first order logic.

- *Structural Properties*
  The synchronized product computes a global synchronization system, in respect with the synhronization constraint. By construction, the synchronization systems are finite and they can be non-deterministic.

- *Behavioural Properties*
  In the ATM case study, there are no deadlock (the final states are not supposed to be blocking) for the `withdrawal` service in the context of the `behaviour` calling service. Obviously, the system has not a liveness property, since it can not ne reinitialized: when we require a service, we hope it to end ! The result of the sequential (resp. parallel) version is detailed in appendix A.1.2 (resp. appendix A.2.2). There are no difference between the two versions because nothing happends between asking the amount and waiting for the amount (state `e6` in the service `withdrawal`).

6.3.4 Inconsistencies Detection

In this section, we arbitrarily introduced inconsistencies to check that MEC detects them. In section 6.3.4 a protocol inconsistency occurs in the `withdrawal` service, then the problem of non-determinism in communications is studied in section 6.3.4.

Behavioural Inconsistency
The model of figure 2 is enriched by a communication related to the card exchange. The `behaviour` service of the `USER_INTERFACE` component of figure 2 is modified from state `e1` to recover the card:

```
...
e1 -- ask_money?recoverCard(c:CashCard) --> e2 ,
                // get the updated card
e2 -- ask_money?result(b:Boolean) --> e3,
      // wait for the result of the withdrawal
...
```

The provided service `withdrawal` of the component `ATM_CORE` of figure 2 is updated to get back the card: a communication replaces the internal action.

```
...
  e3 -- [rep] display("Enter the cash amount, please ?") --> e5,
              // the AAC accepts the transaction, the amount is asked
  e3 -- [not rep] { // the AAC refuses the transaction, the service ends
              display("Transaction refused") ; // calls an internal action
              !recoverCard(c)                  // gets back the card
        }--> e4 ,
...
  e7 -- [m <= card.limit] {
              debit(c,m)          // calls a internal action
              !recoverCard(c)     // gets back the card
              }--> e8 ,
...
```

The straight translation into MEC provides the specification of section A.3.1. When the user recovers his card, the ATM gets back the card. Its execution is given in section A.3.2. The state `(e1,e4)` is locked: the user waits for the card which has been swallowed.

Non determinism and Inconsistencies
We show an inconsistency due to the local non-determinism. Let the (partial) transition systems of figure 6.

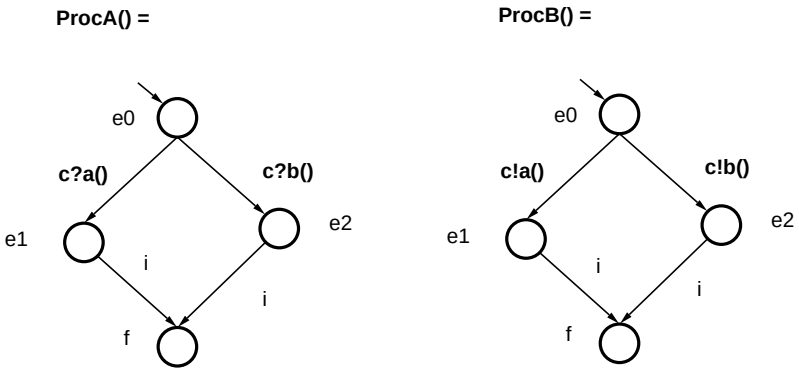

Figure 6 : *Mismatch of Service behaviour*

The straight translation into MEC provides the specification of section B.1.1. Only the consistent communications are allowed, the global synchronization system includes 8 states. The result of the MEC execution is given in section B.1.2. There are no deadlock because the inconsistent communications (e.g. emitting the message `a` on the service `c` and the reception of the message `b` on the service `c`) are ignored by the synchronization constraint.

In order to make the detection of inconsistencies more powerful, we must describe finer communications. For example, the channel (gate, service) can be distinguished from the message sending. In our translation algorithm, it means that a communication is interpreted as a compound sequential transition: select the channel (service) and the mode (in/out), send or receive the message. In such a case, one can detect inconsistent messages on the same channel (emit one message and receive another one). The straight translation into MEC provides the specification of section B.1.3. The result of the MEC execution is given in section B.1.4. MEC detects two erroneous cases (two "deadlocks"):

- sending the message `a` on the service `c` and receiving the message `b` on the service `c`,
- emitting the message `b` on the service `c` and receiving the message `a` on the service `c`.

More inconsistencies are detected, but the translation is more heavy.

Last, one can also enrich the description when considering an additionnal level: channel selection, communication mode choice, message invocation. Having three level communication primitives improves the error detection by adding two error cases[6]: both service emit (or receive) a message.

6.4 Extensions

In this section, we study some concepts of the component model that were not translatable directly in MEC, especially the guards, the value passing and the necessity of multiple instances for one service definition.

6.4.1 Guards

The service definition allows conditions for the transition but MEC does not accept guards nor dynamic evaluation of programs.

A Restricted Semantics for the Guards

A first solution consists in restricting the usage of guards. In section 2.1.2, we assume that for any states, the outgoing transitions labeled by a guard are complentary and exclusive. We cannot check it, but under this asumption, the translation is no more than a guard naming in the current transition translation.

The transition guards are named in a map table and their name are added as prefix to the label string issuing from the current transition translation.

For example, the provided service `withdrawal` of the component `ATM_CORE` of figure 2 is updated to get back the card: a communication replaces the internal action.

```
    ...
      e2 -- [c=card.code] rep := ask_authorization(card.id, c) --> e3 ,
                  // call the required service ask_authorization
      e2 -- [c<>card.code & nbt > 0] display("Enter your card code, please ") --> e0 ,
                  // call an internal action
      e2 -- [c<>card.code & nbt = 0] {
                  display("Card swallowed, sorry") ;   // call an internal action
                  swallow_card()               // call an internal action
            } --> e4 ,
    ...
```

is translated by

```
    ...
e2 |- e -> e2,
     g1_rep_ask_authorization -> e3,
     g2_display -> e0,
     g3_display -> e2_1; \* sequential transition *\
e2_1 |- swallow_card -> e4;
    ...
```

There are no ambiguities.

Static Evaluation of the Guards

To explore the execution traces, a second solution consists in adding a static evaluation for each guard by two values `guardIsTrue` et `guardIsFalse`. This may happend when no asumptions is set on the different guards issuing from the same state. The guards are treated apart. The problem is that in many examples there's a relation between these guards, in our examples `g1`, `g2` and `g3` are exclusive. A solution is to propose a guard language to the specifier to provide a limited but nice set of logic operators (`not`, `and`, `or`) and atomic guard names. The

[6] Which are not considered as errors by LOTOS.

advantage is to check the previous asumptions (determinism) without providing a full language for evaluating expressions (integers, reals, records...).

For example the guards issuing from the state `e2` the provided service `withdrawal` of the component `ATM_CORE` of figure 2 are named as follows:

```
g_e2_1 == (c = carte.code)
g_e2_2 == (nbe > 0)
```

Note that in the last transition, the initial specification is complete only if `nbe` is a natural number.

The transitions are written as follow:

```
    ...
      e2 -- [g_e2_1] rep := ask_authorization(card.id, c) --> e3 ,
                  // calls the required service ask_authorization
      e2 -- [(not g_e2_1) and g_e2_2] display("Enter your card code, please ") --> e0 ,
                  // calls an internal action
      e2 -- [(not g_e2_1) and (not g_e2_2)] {
                  display("Card swallowed, sorry") ;   // calls an internal action
                  swallow_card()               // calls an internal action
            } --> e4 ,
    ...
```

The translation into MEC is twofold:

1. adding explicit transitions for evaluating the guards first,
2. prefixing the transition labels by the guard expressions.

The problem with the first solution is that it can introduce deadlocks. For example, a guard can be true but the associated communication action fails i.e. a message send cannot be send because there are no reception in the corresponding service.

The above example is translated into

```
    ...
e2 |- e -> e2,
     g_e2_1_rep_ask_authorization -> e3,
     not_g_e2_1_and_g_e2_2_display -> e0,
     not_g_e2_1_and_not_g_e2_2_display -> e2_1; \* sequential transition *\
e2_1 |- swallow_card -> e4;
    ...
```

This work needs further experimentations. One issue is to use a symbolic model checker (for example, MEC 5 includes guards on state values, but not on parameters). Another issue is to combine the guard evaluation with the tool used for verifying the assertions (pre/post conditions). An experimentation with B [1] is under study.

Delayed Guard Handling by Removing Unreachable States

The unreacheable deadlock states are the (reachable) states of the synchronized product that cannot be reached when taking the semantics of the guards into account. The translation algorithm cannot avoid them, but the verification process can handle them interactively (the specifier decides which deadlock is a true or false deadlock).

```
write or modify the service specification (1)
translate into MEC (2)
unreachable_deads := empty  /* stores the unreacheable deadlock states */
true_deads := empty         /* stores the reacheable deadlock states */
repeat
  deadlock := run the MEC evaluation
  no_more_unreachable_deads := true
```

```
  for each s : State in (deadlock − unreachable_deads) do
    answer := read("Is state ", s, " an unreachable deadlock
                    when considering the guards ?")
    if answer then
        unreachable_deads := add(s, unreachable_deads) ;
        /* s becomes invalid for the verification */
        no_more_unreachable_deads := false
    else
        true_deads := add(s, true_deads)
    endif
  endfor
until no_more_unreachable_deads ; /* fixpoint */
```

The above ad-hoc solution is a post specification processing since it does not change the initial specification.

Sometimes several unreachable deadlock states share a same property. Thus, in order to treat them collectively (to avoid an individual processing), one can mark the LTS states directly in the MEC specification.

Note that the steps (1) and (2) can be set inside the repeat loop to remove the true deadlocks too.

```
unreachable_deads := empty
repeat
  write or modify the service specification (1)
  translate into MEC (2)
  deadlock := run the MEC evaluation
  for each s : State in (deadlock − unreachable_deads) do
    answer := read("Is state ", s, " an unreachable deadlock
                    when considering the guards ?")
    if answer then
       unreachable_deads := add(s, unreachable_deads)
       /* s becomes invalid for the verification */
    endif
  endfor
until deadlock = unreachable_deads ; /* fixpoint */
```

This work needs further experimentations.

6.4.2 Managing Communications and Parameters

The parameters are used for the guards and for the component state changes. Since no symbolic evaluation is done for this cases, the management of parameters has no sense in MEC.

6.4.3 Multiple Instances of a Service

Until now, each service is instanciated once: each service terminates (on final states). What happens for multiple instance of service ?

In a first approach, we can consider several sequential instanciations: that means that each service has a unique instance at any time, but several instances during the program execution. Such a recurrent service call is easily implemented by reinitialization transitions (the final state becomes an initial state).

Handling multiple instances of service requires the naming of on-going service and in somewhat implementation a scheduler for running them. In MEC, each (running) service is represented by a column in the synchronization constraint. Thus all the invocable services are statically known (no dynamic creation, no recursive service call). The several instances of one service are distinguished by their column number. This implicit identification has to be maintained by the MEC translator to keep the consistency between the service interaction.

This problem is still open in the Kmelia component model.

6.4.4 Composition

MEC 4 does not allow hierachical composition of LTS: there are two levels LTS and systems. Since systems are not LTS, no composition is possible. We think that MEC could be easily extended to composition by naming the lines of the synchronization constraint without theoretical problems.

6.4.5 AltaRica

The Altarica tool [8] uses a new release of MEC, called MEC 5. MEC 5 is implemented by Binary Decision Diagrams (BDD), one of the symbolic model checking techniques. According to the authors, MEC 4 specifications can be inputs of MEC 5 but "nothing useful can be extracted from them with MEC 5 yet".

Altarica is both a dataflow language and a state transition system. We assume the second interpretation. The states are not explicit names (as in MEC 4) but they are the (implicit) cartesian product of state variables, in which types are restricted to integers, booleans and enumerated types. The Altarica guards are conditions on these state variables so that the source state of a transition is the result of guard expression. Of course a MEC 4 LTS can be translated into Mec 5 by a single state variable, in which type is an enumerated type. Altarica transitions have no parameter.

This short description shows that the above extensions cannot be simply improved by AltaRica: no parameters, no powerful usage of Mec 5 states, reduced guards application. Thus the generated specification would be far from the service behaviour and less readable.

6.5 Implementation

The above ideas have been inplemented in our automated environment called `COSTO`. This specification environment currently accepts component specification and handled various functionalities on the resulting code. It is implemented in ANTLR and Java.

- The `Kml2mec` class implements the translator. The translator accepts as input a Kmelia description of a composition together with a service context. It generates a Mec file.
- The Mec file is evaluated under Mec 4, using the `load(file);` command. It generates a model checking result file.
- The `Mec2Dot` class implements a translator from MEC 4 model checking result file into a Dot file (for visual description).

We experimented the sequential approach on the ATM case study. The Mec translation operates in the following context :

- service to check : `withdrawal`
- calling service : `behaviour`
- required service : `ask_for_money`

There are four LTS one per service and one for the provided services (`amount`, `id`) that are called by required subservices. The final STS includes 1153 states and 2669 transitions. There are no deadlocks, the triple `withdrawal-behaviour-ask_for_money` is therefore behaviourally consistent.

6.6 Conclusion and perspectives

Since Kmelia is more powerful than MEC, some Kmelia features cannot map to MEC concepts *e.g.* guards, parameters, results... Nevertheless, MEC asserts a first (basic) level of behavioural compatibility. MEC ensures a first confidence level, when the constraints on the symbolic aspects of the specification are lightened (no guard and parameters). Other experiments with state transition system could be led using other model checker such as SPIN, or PVS, or with other theories like the π-calculus.

7 An overview of The COSTO Toolbox

Our proposal is made with the sake of mechanization. We start the development of a prototype named COSTO (Component Study Toolbox) to support all the steps of component analysis and developement.

The prototype already integrates:

- the Kmelia specification parsers,
- a translator to LOTOS,
- a translator to MEC,
- the static interoperability checkers,
- the dynamic interoperability checkers,
- a translator of Kmelia services into dot (for the visualisation of service behaviours).

However we lack a graphical interface to guide and assist the user. This is subject to current developement.

The figure 7 gives an overview of the main parts of the COSTO toolbox.

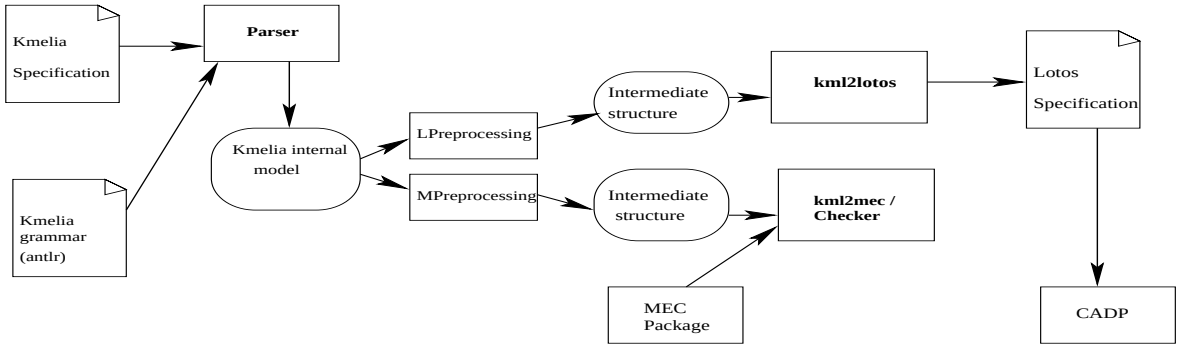

Figure 7: An overview of the COSTO

8 Related Work and Discussion

We have presented an abstract component model and a formalism (Kmelia) that permits the flexible description of interacting services which are defined as extended labelled transition systems.
We have proposed a behavioural verification algorithm to check the compatibility between service behaviours.

From the practical point of view, our proposal follows the mechanized approaches like Darwin/Tracta [22, 17] or SOFA [27].
Darwin [22] is an architecture description language that describes systems in terms of components. The latter ones manage the implementation of services. A system has an hierarchical structure including primitive components and composites. The primitive components are equipped with a behaviour expressed as a labelled transition system. A composite behaviour is the parallel composition of the primitive behaviour.
Tracta [17] is a compositional reachability analysis approach; it is used in combination with Darwin to analyse distributed systems. The component behaviours are described with finite state processes.

In the SOFA component model[26], components are either primitive or composed. A primitive component contains functionality (described as methods) and no subcomponents. A composed component contains subcomponents and no functionality. The methods of a component is used in various forms called events (emission, recepetion and their results) SOFA components are linked with connectors. A SOFA component has a behaviour protocol defined with a regular-like expression made of the events built using the component methods.

The Fractal Component Model [14] is an extensible component model dedicated to design, implementation and deployment. It can be used used with various programming languages, systems, applications. In the Fractal approach a component has a controller that gives the bahaviour of the particular component. Therefore our proposal is different of Fractal.

Unlike most of existing approaches [35, 26, 9, 13] where a component has a behaviour, we argue for a model were the provided services defined as LTS have behaviour. This moves up the granularity for the use of components and increases the usability of components by considering service level. When our service behaviours are reduced to combinations of messages, we get the low level of usability found in the aforementioned approaches.
In our approach, the composition of the services is the support of component compositions; it is performed by considering their interfaces and their behaviours. We can extract several collaborations à la Yellin&Strom [35] from a single of our service behaviours which interweaves collaborations on different channels and allows optional calls of services.
At the behavioural verification level, the union a the collaboration à la Yellin&Strom is a subset of the possible interactions that we can verify.

Unlike the works presented in [9, 20], our approach works at the abstract specification level; it offers a more flexible formalism than the ones proposed by [35, 9] for the description of interacting services. We adopt a pairwise verification approach that avoids state explosion like in [9].
Tracta and SOFA already provide many analysis tools; but we have a different component model that needs deep investigation before tool reuse and development. However we can build on the experiences gained with these works.

Most of the approaches that integrate behavioural specifications to components [27, 25, 31] work at a protocol (or component) level while our approach is mainly on the services, the protocol level is handled by a constraint in our model. Moreover, their communication actions refer only to messages and not to services (no service call or result). The non-regular protocols of [31] can be represented in Kmelia using guards and nested states, but using algebraic grammar provides a more efficient solution for the given applications. The work of [25] addresses assemblies and implementation issues in Java but does not deal with composition.

The study of compatibility at the component behaviour level is central to CBSE approaches and motivated number of works [35, 15, 9, 13] and application to web-services [12]. We build on these works but we extend the study to encompass the granularity considered here for services and components. Our approach allows for a local verification of the behavioural compatibility between composed services. Experiments are performed with the approach using existing toolboxes (Lotos CADP and MEC).

9 Conclusion and Perspectives

We have presented a formal abstract component model based on services. These latter are described through their behavioural specifications using extended labelled transition systems. We have not deal with *component based programming* and the related models such as EJB, .net, Corba Component Model, etc.

We proposed a behavioural verification algorithm to check the compatibility between service behaviours. The algorithm is adapted to the parallel composition and communication offered by Lotos and Mec in order to conduct rapid experimentations. An ATM example is used to illustrate the proposed approach along the report .

We have proposed a simple formalism to describe components and their composition. A general composition operator is defined to build large systems from simple components. For the simplicity of the model and the proposed component specification formalism, the model uses only components, there is no connector as in the other approaches [29, 24]. We then have an homogeneous treatment of component compositions. The main features of our proposal are: simplicity and homogenity. Indeed our basic structure is the service (which is accessible via the components); we defined operators to handle services (via the assemblies or the compositions).

An experimental framework is associated to this model so as to study various related aspects: interface compatibility, behavioural compatibility, correctness (safety, liveness) and maintenance.

A component is formalised with a model-based state space which may be modified with a set of labelled transition systems that describe the services. Only one service is active at time. The services communicate synchronously or asynchronously.

We investigate a set of formal analysis topics for the study of components. We specifically focus on behavioural conformity analysis. We elaborate an interaction verification algorithm based on consistency rules.

We conduct two main experimentations for mechanizing the analysis of the interaction between components. One experimentation is achieved using the LOTOS/CADP framework. We systematically translate the services behaviour into LOTOS processes and then the CADP tools are used to check the interaction between the processes. CADP provides several powerful tools to analyse processes. The current results are quite satisfactory; we can detect interaction flaws.

The second experimentation is achieved in a similar way: the component services are systematically translated into the MEC/Altarica framework. The description in MEC is more user friendly but MEC is less powerful with respect to the need of services.

Perspectives. Many exciting investigations remain to do. Whatever the component model, the compositionality is still a challenge [34].

Ongoing works cover: the extension of our component description language, the implementation of some translators (into LOTOS and MEC) in order to support the mechanization with the related tools.

More generally we are building a component study toolbox (COSTO) to support the full processof modeling, analysis and implementation of components. COSTO is a framework that already integrates some parsers and some analysers which generate labelled transition systems. We are working on the process generators to build the bridges with the external tools (CADP and MEC).

The concrete plan for the perspectives of the current work is the following:

- the extension of the bridges with existing component frameworks and formal COSTO analysis toolbox; the prototype already integrates parsers, translators to LOTOS and MEC, static and dynamic interoperability checkers. However we lack a graphical interface to guide and assist the user. Then we will propose an open source delivery of the toolbox.
- the enhancement of the dynamic aspects of component analysis (behavioural interaction).
- the construction of concrete components from the abstract ones: exploiting refinement techniques;
- the study of the heterogeneity of components (how to treat properly the components coming from various providers).
- some experiments with the B method (theorem proving correctness properties) are also envisaged; we are investigating the verification of functional properties of services,
- to reinforce the correctness properties of component with supplementary study of correctness of components and services with regard to their environment.

References

[1] Jean-Raymond Abrial. *The B-Book Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

[2] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[3] Pascal André and Alain Vailly. *Conception de systèmes d'information, Panorama des méthodes et des techniques*, volume 1 of *Collection Technosup*. Editions Ellipses, 2001. ISBN 2-7298-0479-X.

[4] P. André, G. Ardourel, and C. Attiogbé. Behavioural Verification of Service Composition. In *ICSOC 2005 Workshop on Engineering Service Compositions*, 2005.

[5] P. André, G. Ardourel, C. Attiogbé, H. Habrias, and C. Stoquer. Vérification de conformité des interactions entre composants. Technical report RR04.11, LINA - FRE CNRS 2729 - Nantes, December 2004.

[6] André Arnold. *Systèmes de transitions finis et sémantique des processus communiquants*. Collection Etudes et recherches en informatique. Masson, 1992. ISBN 2-225-82746-X.

[7] André Arnold, Paul Crubillé, and Didier Bégay. *Construction and Analysis of Transition Systems with MEC*. AMAST Series in Computing: Vol. 3. World Scientific, 1994. ISBN 981-02-1922-9.

[8] André Arnold, Alain Griffault, Gérald Point, and Antoine Rauzy. The altarica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40:109–124, 2000.

[9] P. C. Attie and D. H. Lorenz. Correctness of Model-based Component Composition without State Explosion. In *ECOOP 2003 Workshop on Correctness of Model-based Software Composition*, 2003.

[10] P. C. Attie and D. H. Lorenz. Establishing Behavioral Compatibility of Software Components without State Explosion. Technical report NU-CCIS-03-02, College of Computer and Information Science, Northeastern University, 2003.

[11] K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy. A Formal Model for Componentware. In G. T. Leavens and M. Sitaraman, eds., *Foundations of Component-Based Systems*, pages 189–210. Cambridge University Press, New York, NY, 2000.

[12] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In *TES*, pages 15–28, 2004.

[13] A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.

[14] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *ECOOP 2002 Workshop on Component-Oriented Programming (WCOP02, Spain)*, 2002.

[15] L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.

[16] J-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In R. Alur and T. A. Henzinger, eds., *Proc. of the 8th Conference on Computer-Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 437–440. Springer Verlag, 1996.

[17] D. Giannakopoulou, J. Kramer, and S.C. Cheung. Behaviour Analysis of Distributed Systems Using the Tracta Approach. *Autom. Softw. Eng.*, 6(1):7–35, 1999.

[18] T. Gschwind, U. Aßmann, and O. Nierstrasz, eds.. *Software Composition, 4th Int. Workshop, SC 2005, Edinburgh, UK*, volume 3628 of *Lecture Notes in Computer Science*. Springer, 2005.

[19] G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, eds.. *Component-Based Software Engineering, 8th International Symposium, CBSE 2005, USA, May, 2005*, volume 3489 of *LNCS*. Springer, 2005.

[20] P. Inverardi, A. L. Wolf, and D. Yankelevich. Static Checking of System Behaviors using Derived Component Assumptions. *ACM Transactions on Software Engineering and Methodology*, 9(3):239–272, 2000.

[21] ISO LOTOS. *A Formal Description Technique Based on The Temporal Ordering of Observational Behaviour*. International Organisation for Standardization - Information Processing Systems - Open Systems Interconnection, Geneva, 1988.

[22] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, eds., *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[23] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, january 2000.

[24] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 178–187. ACM Press, 2000.

[25] S. Pavel, J. Noyé, P. Poizat, and J.C. Royer. A Java Implementation of a Component Model with Explicit Symbolic Protocols. In GSCHWINDet al. [18], pages 115–124.

[26] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In *ICCDS'98, IEEE CS Press*, 1998.

[27] F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on SW Engineering*, 28(9), 2002.

[28] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall International, 1991.

[29] M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice Hall, 1996.

[30] Mike Spivey. *The Z notation: a Reference Manual*. International Series in Computer Science. Prentice-Hall, 1989.

[31] M. Südholt. A Model of Components with Non-regular Protocols. In GSCHWINDet al. [18], pages 99–113.

[32] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. AddisonWesley Publishing Company, 1997.

[33] A. Vaillant and S. Rodier. Vérification de la conformité des interactions entre composants. Technical report TER M1.2004/2005, LINA - FRE CNRS 2729 - Nantes, 2005.

[34] F. Xie and J. C. Browne. Verified Systems by Composition from Verified Components. In *ESEC/FSE-11: Proc. of the 9th European software engineering conference*, pages 277–286, New York, NY, USA, 2003. ACM Press.

[35] D.M. Yellin and R.E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

A The ATM Case Study in MEC

This appendix includes the MEC specifications and results for the ATM Case Study.

A.1 Sequential System

A.1.1 Specification

```
\*
      Author: P. ANDRE
      Project: COLOSS
      Experimentation : model-checking of services with MEC
      Case study : service Retrait du BASEGAB

      Version 1 : Systeme sequentiel sans gardes et parametres

      Creation date: 05/04/05
      Modification date: 05/04/05

      load(gabretraitSeq);
   *\

   transition_system IHM_CLIENTR_DemRetrait < width = 0 >;
   init |- e -> init,
        start_DemRetrait -> e0;
   e0 |- e -> e0,
        call_Retrait -> e1; \* transition d'initialisation *\
   e1 |- e -> e1,
        start_Code -> e1_1, \* inclusion du service Code *\
        start_Montant -> e1_2, \* inclusion du service Montant *\
        rcv_Retrait_resRetrait -> e2; \* e2 est suppose final *\
   e1_1 |- e -> e1_1, \* developpement du service Code *\
        emit_Code_resCode -> e1;
   e1_2 |- e -> e1_2, \* developpement du service Montant *\
        saisie_sel -> e1_3;
   e1_3 |- e -> e1_3,
        emit_Montant_resMontant -> e1;
   e2 |- e -> e2;
   < initial = { init } ; final = {e2} >.

   transition_system BASEGAB_Retrait < width = 0 >;
   init |- e -> init, \* transition d'initialisation *\
        start_Retrait -> i;
   i |- e -> i,
        nbe3 -> e0;
   e0 |- e -> e0,
        call_Code -> e1;
   e1 |- e -> e1,
        rcv_Code_resCode -> e1_1; \* transition sequentielle *\
   e1_1 |- nbe_nbe_1 -> e2;
   e2 |- e -> e2,
        msg -> e0,
        rep_VerifAuth -> e3,
        msg -> e2_1; \* transition sequentielle *\
   e2_1 |- avalerCarte -> e4;
   e3 |- e -> e3,
        msg -> e5,
        msg -> e3_1; \* transition sequentielle *\
   e3_1 |- restituerCarte -> e4;
   e4 |- e -> e4,
        emit_Retrait_resRetrait -> f;
   e5 |- e -> e5,
        call_Montant -> e6;
   e6 |- e -> e6,
```

```
     rcv_Montant_resMontant -> e7;
e7 |- e -> e7,
     msg -> e3,
     debiter -> e7_1; \* transition sequentielle *\
e7_1 |- restituerCarte -> e8;
e8 |- e -> e7,
     emit_Retrait_resRetrait -> f;
< initial = { init } ; final = {f} >.

synchronization_system Verif_BASEGAB_Retrait < width = 2 ;
    list = ( IHM_CLIENTR_DemRetrait, BASEGAB_Retrait ) >;

\* action relative au lancement du service appelant *\
( start_DemRetrait . e );
\* actions interne de l'automate IHM_CLIENTR_DemRetrait *\
( saisie_sel . e );
\* actions interne de l'automate BASEGAB_Retrait *\
( e . nbe3 );
( e . nbe_nbe_1);
( e . msg );
( e . rep_VerifAuth );
( e . avalerCarte );
( e . restituerCarte );
( e . debiter);
\* invocations de services *\
( call_Retrait . start_Retrait );
( start_Code . call_Code );
( start_Montant . call_Montant );
\* communications sur Retrait *\
( rcv_Retrait_resRetrait . emit_Retrait_resRetrait );
\* communications sur Code *\
( emit_Code_resCode . rcv_Code_resCode );
\* communications sur Montant *\
( emit_Montant_resMontant . rcv_Montant_resMontant ).

function inevitable(Y:trans ; X:state) return Z:state;
begin
Z = X \/ (src(Y /\ rtgt(Z)) - src(Y /\ rtgt(*-Z)))
end.

sync(Verif_BASEGAB_Retrait, Verif_BASEGAB_Retrait);
dts(Verif_BASEGAB_Retrait);
finaux := final[1] /\ final[2];
dead := (* - src(*)) - finaux;
deadlock:=inevitable(*,dead);
boucle := loop(*,*);
log(gabretraitSeqLog);
version();
dts(Verif_BASEGAB_Retrait);
wts(*,*);
stoplog();
```

A.1.2 Results

```
version();
   //MEC 4 version alpha 3.4dts(Verif_BASEGAB_Retrait);
   //Current St : Verif_BASEGAB_Retrait
   //-- States :
   // * : 18
   //initial : 1
   //finaux : 1
   //dead : 0
   //deadlock : 0
   //-- Transitions :
   // * : 21
```

```
//boucle : 9
//wts(*,*);
//transition_system Verif_BASEGAB_Retrait
//< width = 2;  list = (IHM_CLIENTR_DemRetrait, BASEGAB_Retrait )>;
//
//
//e(init.init) |-
// (start_DemRetrait.e) -> e(e0.init);
//
//e(e0.init) |-
// (call_Retrait.start_Retrait) -> e(e1.i);
//
//e(e1.i) |-
// (e.nbe3) -> e(e1.e0);
//
//e(e1.e0) |-
// (start_Code.call_Code) -> e(e1_1.e1) < property = ( boucle  ) >;
//
//e(e1_1.e1) |-
// (emit_Code_resCode.rcv_Code_resCode) -> e(e1.e1_1) < property = ( boucle  ) >;
//
//e(e1.e1_1) |-
// (e.nbe_nbe_1) -> e(e1.e2) < property = ( boucle  ) >;
//
//e(e1.e2) |-
// (e.msg) -> e(e1.e0) < property = ( boucle  ) >,
// (e.rep_VerifAuth) -> e(e1.e3),
// (e.msg) -> e(e1.e2_1);
//
//e(e1.e3) |-
// (e.msg) -> e(e1.e5) < property = ( boucle  ) >,
// (e.msg) -> e(e1.e3_1);
//
//e(e1.e2_1) |-
// (e.avalerCarte) -> e(e1.e4);
//
//e(e1.e5) |-
// (start_Montant.call_Montant) -> e(e1_2.e6) < property = ( boucle  ) >;
//
//e(e1.e3_1) |-
// (e.restituerCarte) -> e(e1.e4);
//
//e(e1.e4) |-
// (rcv_Retrait_resRetrait.emit_Retrait_resRetrait) -> e(e2.f);
//
//e(e1_2.e6) |-
// (saisie_sel.e) -> e(e1_3.e6) < property = ( boucle  ) >;
//
//e(e2.f) |- ;
//
//e(e1_3.e6) |-
// (emit_Montant_resMontant.rcv_Montant_resMontant) -> e(e1.e7) < property = ( boucle  ) >;
//
//e(e1.e7) |-
// (e.msg) -> e(e1.e3) < property = ( boucle  ) >,
// (e.debiter) -> e(e1.e7_1);
//
//e(e1.e7_1) |-
// (e.restituerCarte) -> e(e1.e8);
//
//e(e1.e8) |-
// (rcv_Retrait_resRetrait.emit_Retrait_resRetrait) -> e(e2.f);
//<
//initial = { e(init.init) },
//finaux = { e(e2.f) },
```

```
//dead = {  },
//deadlock = {  }
//>.
//stoplog();
//
```

A.2 Parallel System

A.2.1 Specification

```
\*
      Author: P. ANDRE
      Project: COLOSS
      Experimentation : model-checking of services with MEC
      Case study : service Retrait du BASEGAB

      Version 1 : Systeme sequentiel sans gardes et parametres

      Creation date: 05/04/05
      Modification date: 05/04/05

      load(gabretraitPar);
   *\

   transition_system IHM_CLIENTR_DemRetrait < width = 0 >;
   init |- e -> init,
        start_DemRetrait -> e0;
   e0 |- e -> e0,
        call_Retrait -> e1; \* transition d'initialisation *\
   e1 |- e -> e1,
        start_Code -> e1_1, \* inclusion du service Code *\
        start_Montant -> e1_2, \* inclusion du service Montant *\
        rcv_Retrait_resRetrait -> e2; \* e2 est suppose final *\
   e1_1 |- e -> e1_1, \* developpement du service Code *\
        emit_Code_resCode -> e1;
   e1_2 |- e -> e1_2, \* developpement du service Montant *\
        saisie_sel -> e1_3;
   e1_3 |- e -> e1_3,
        emit_Montant_resMontant -> e1;
   e2 |- e -> e2;
   < initial = { init } ; final = {e2} >.

   transition_system BASEGAB_Retrait < width = 0 >;
   init |- e -> init, \* transition d'initialisation *\
        start_Retrait -> i;
   i |- e -> i,
        nbe3 -> e0;
   e0 |- e -> e0,
        call_Code -> e1;
   e1 |- e -> e1,
        rcv_Code_resCode -> e1_1; \* transition sequentielle *\
   e1_1 |- nbe_nbe_1 -> e2;
   e2 |- e -> e2,
        msg -> e0,
        rep_VerifAuth -> e3,
        msg -> e2_1; \* transition sequentielle *\
   e2_1 |- avalerCarte -> e4;
   e3 |- e -> e3,
        msg -> e5,
        msg -> e3_1; \* transition sequentielle *\
   e3_1 |- restituerCarte -> e4;
   e4 |- e -> e4,
        emit_Retrait_resRetrait -> f;
   e5 |- e -> e5,
        call_Montant -> e6;
```

```
e6 |- e -> e6,
     rcv_Montant_resMontant -> e7;
e7 |- e -> e7,
     msg -> e3,
     debiter -> e7_1; \* transition sequentielle *\
e7_1 |- restituerCarte -> e8;
e8 |- e -> e7,
     emit_Retrait_resRetrait -> f;
< initial = { init } ; final = {f} >.

synchronization_system Verif_BASEGAB_Retrait < width = 2 ;
    list = ( IHM_CLIENTR_DemRetrait, BASEGAB_Retrait ) >;

\* action relative au lancement du service appelant *\
( start_DemRetrait . e );
\* actions interne de l'automate IHM_CLIENTR_DemRetrait *\
( saisie_sel . e );
\* actions interne de l'automate BASEGAB_Retrait *\
( e . nbe3 );
( e . nbe_nbe_1);
( e . msg );
( e . rep_VerifAuth );
( e . avalerCarte );
( e . restituerCarte );
( e . debiter);
\* invocations de services *\
( call_Retrait . start_Retrait );
( start_Code . call_Code );
( start_Montant . call_Montant );
\* communications sur Retrait *\
( rcv_Retrait_resRetrait . emit_Retrait_resRetrait );
\* communications sur Code *\
( emit_Code_resCode . rcv_Code_resCode );
\* communications sur Montant *\
( emit_Montant_resMontant . rcv_Montant_resMontant );
\* --------------- *\
\* 2 actions en parallèles *\
\* --------------- *\
( saisie_sel . nbe3 );
( saisie_sel . nbe_nbe_1);
( saisie_sel . msg );
( saisie_sel . rep_VerifAuth );
( saisie_sel . avalerCarte );
( saisie_sel . restituerCarte );
( saisie_sel . debiter).

function inevitable(Y:trans ; X:state) return Z:state;
begin
Z = X \/ (src(Y /\ rtgt(Z)) - src(Y /\ rtgt(*-Z)))
end.

sync(Verif_BASEGAB_Retrait, Verif_BASEGAB_Retrait);
dts(Verif_BASEGAB_Retrait);
finaux := final[1] /\ final[2];
dead := (* - src(*)) - finaux;
deadlock:=inevitable(*,dead);
boucle := loop(*,*);
log(gabretraitParLog);
version();
dts(Verif_BASEGAB_Retrait);
wts(*,*);
stoplog();
```

A.2.2 Results

```
version();
   //MEC 4 version alpha 3.4dts(Verif_BASEGAB_Retrait);
   //Current St : Verif_BASEGAB_Retrait
   //-- States :
   // * : 18
   //initial : 1
   //finaux : 1
   //dead : 0
   //deadlock : 0
   //-- Transitions :
   // * : 21
   //boucle : 9
   //wts(*,*);
   //transition_system Verif_BASEGAB_Retrait
   //< width = 2;  list = (IHM_CLIENTR_DemRetrait, BASEGAB_Retrait )>;
   //
   //
   //e(init.init) |-
   // (start_DemRetrait.e) -> e(e0.init);
   //
   //e(e0.init) |-
   // (call_Retrait.start_Retrait) -> e(e1.i);
   //
   //e(e1.i) |-
   // (e.nbe3) -> e(e1.e0);
   //
   //e(e1.e0) |-
   // (start_Code.call_Code) -> e(e1_1.e1) < property = ( boucle  ) >;
   //
   //e(e1_1.e1) |-
   // (emit_Code_resCode.rcv_Code_resCode) -> e(e1.e1_1) < property = ( boucle  ) >;
   //
   //e(e1.e1_1) |-
   // (e.nbe_nbe_1) -> e(e1.e2) < property = ( boucle  ) >;
   //
   //e(e1.e2) |-
   // (e.msg) -> e(e1.e0) < property = ( boucle  ) >,
   // (e.rep_VerifAuth) -> e(e1.e3),
   // (e.msg) -> e(e1.e2_1);
   //
   //e(e1.e3) |-
   // (e.msg) -> e(e1.e5) < property = ( boucle  ) >,
   // (e.msg) -> e(e1.e3_1);
   //
   //e(e1.e2_1) |-
   // (e.avalerCarte) -> e(e1.e4);
   //
   //e(e1.e5) |-
   // (start_Montant.call_Montant) -> e(e1_2.e6) < property = ( boucle  ) >;
   //
   //e(e1.e3_1) |-
   // (e.restituerCarte) -> e(e1.e4);
   //
   //e(e1.e4) |-
   // (rcv_Retrait_resRetrait.emit_Retrait_resRetrait) -> e(e2.f);
   //
   //e(e1_2.e6) |-
   // (saisie_sel.e) -> e(e1_3.e6) < property = ( boucle  ) >;
   //
   //e(e2.f) |- ;
   //
   //e(e1_3.e6) |-
   // (emit_Montant_resMontant.rcv_Montant_resMontant) -> e(e1.e7) < property = ( boucle  ) >;
   //
```

```
   //e(e1.e7) |-
   // (e.msg) -> e(e1.e3) < property = ( boucle  ) >,
   // (e.debiter) -> e(e1.e7_1);
   //
   //e(e1.e7_1) |-
   // (e.restituerCarte) -> e(e1.e8);
   //
   //e(e1.e8) |-
   // (rcv_Retrait_resRetrait.emit_Retrait_resRetrait) -> e(e2.f);
   //<
   //initial = { e(init.init) },
   //finaux = { e(e2.f) },
   //dead = {  },
   //deadlock = {  }
   //>.
   //stoplog();
   //
```

A.3 Inconsistent System

A.3.1 Specification

```
\*
      Author: P. ANDRE
      Project: COLOSS
      Experimentation : model-checking of services with MEC
      Case study : service Retrait du GAB

      Version 1r : Systeme sequentiel sans gardes et parametres
                   avec erreur

      Creation date: 05/04/05
      Modification date: 11/04/05

      load(gabretraitSeqErr);
   *\

   transition_system IHM_CLIENTR_DemRetrait < width = 0 >;
   init |- e -> init,
        start_DemRetrait -> e0;
   e0 |- e -> e0,
        call_Retrait -> e1; \* transition d'initialisation *\
   e1 |- e -> e1,
        start_Code -> e1_1, \* inclusion du service Code *\
        start_Montant -> e1_2, \* inclusion du service Montant *\
        rcv_Retrait_recupCarte -> e2;
   e1_1 |- e -> e1_1, \* developpement du service Code *\
        emit_Code_resCode -> e1;
   e1_2 |- e -> e1_2, \* developpement du service Montant *\
        saisie_sel -> e1_3;
   e1_3 |- e -> e1_3,
        emit_Montant_resMontant -> e1;
   e2 |- e -> e2, \* fin du service Retrait *\
        rcv_Retrait_resRetrait -> e3;
   e3 |- e -> e3; \* e3 est suppose final *\
   < initial = { init } ; final = {e3} >.

   transition_system GAB_Retrait < width = 0 >;
   init |- e -> init, \* transition d'initialisation *\
        start_Retrait -> i;
   i |- e -> i,
        nbe3 -> e0;
   e0 |- e -> e0,
        call_Code -> e1;
   e1 |- e -> e1,
```

```
     rcv_Code_resCode -> e1_1; \* transition sequentielle *\
e1_1 |- nbe_nbe_1 -> e2;
e2 |- e -> e2,
     msg -> e0,
     rep_VerifAuth -> e3,
     msg -> e2_1; \* transition sequentielle *\
e2_1 |- avalerCarte -> e4;
e3 |- e -> e3,
     msg -> e5,
     msg -> e3_1; \* transition sequentielle *\
e3_1 |- emit_Retrait_restituerCarte -> e4;
e4 |- e -> e4,
     emit_Retrait_resRetrait -> f;
e5 |- e -> e5,
     call_Montant -> e6;
e6 |- e -> e6,
     rcv_Montant_resMontant -> e7;
e7 |- e -> e7,
     msg -> e3,
     debiter -> e7_1; \* transition sequentielle *\
e7_1 |- emit_Retrait_restituerCarte -> e8;
e8 |- e -> e7,
     emit_Retrait_resRetrait -> f;
< initial = { init } ; final = {f} >.

synchronization_system Verif_GAB_Retrait < width = 2 ;
    list = ( IHM_CLIENTR_DemRetrait, GAB_Retrait ) >;

\* action relative au lancement du service appelant *\
( start_DemRetrait . e );
\* actions interne de l'automate IHM_CLIENTR_DemRetrait *\
( saisie_sel . e );
\* actions interne de l'automate GAB_Retrait *\
( e . nbe3 );
( e . nbe_nbe_1);
( e . msg );
( e . rep_VerifAuth );
( e . avalerCarte );
( e . debiter);
\* invocations de services *\
( call_Retrait . start_Retrait );
( start_Code . call_Code );
( start_Montant . call_Montant );
\* communications sur Retrait *\
( rcv_Retrait_resRetrait . emit_Retrait_resRetrait );
( rcv_Retrait_recupCarte . emit_Retrait_restituerCarte );
\* communications sur Code *\
( emit_Code_resCode . rcv_Code_resCode );
\* communications sur Montant *\
( emit_Montant_resMontant . rcv_Montant_resMontant ).

function inevitable(Y:trans ; X:state) return Z:state;
begin
Z = X \/ (src(Y /\ rtgt(Z)) - src(Y /\ rtgt(*-Z)))
end.

sync(Verif_GAB_Retrait, Verif_GAB_Retrait);
dts(Verif_GAB_Retrait);
finaux := final[1] /\ final[2];
dead := (* - src(*)) - finaux;
deadlock:=inevitable(*,dead);
boucle := loop(*,*);
log(gabretraitSeqErrLog);
version();
dts(Verif_GAB_Retrait);
```

```
   wts(*,*);
   stoplog();
```

A.3.2 Results

```
version();
   //MEC 4 version alpha 3.4dts(Verif_GAB_Retrait);
   //Current St : Verif_GAB_Retrait
   //-- States :
   // * : 19
   //initial : 1
   //finaux : 1
   //dead : 1
   //deadlock : 2
   //-- Transitions :
   // * : 21
   //boucle : 9
   //wts(*,*);
   //transition_system Verif_GAB_Retrait
   //< width = 2;  list = (IHM_CLIENTR_DemRetrait, GAB_Retrait )>;
   //
   //
   //e(init.init) |-
   // (start_DemRetrait.e) -> e(e0.init);
   //
   //e(e0.init) |-
   // (call_Retrait.start_Retrait) -> e(e1.i);
   //
   //e(e1.i) |-
   // (e.nbe3) -> e(e1.e0);
   //
   //e(e1.e0) |-
   // (start_Code.call_Code) -> e(e1_1.e1) < property = ( boucle  ) >;
   //
   //e(e1_1.e1) |-
   // (emit_Code_resCode.rcv_Code_resCode) -> e(e1.e1_1) < property = ( boucle  ) >;
   //
   //e(e1.e1_1) |-
   // (e.nbe_nbe_1) -> e(e1.e2) < property = ( boucle  ) >;
   //
   //e(e1.e2) |-
   // (e.msg) -> e(e1.e0) < property = ( boucle  ) >,
   // (e.rep_VerifAuth) -> e(e1.e3),
   // (e.msg) -> e(e1.e2_1);
   //
   //e(e1.e3) |-
   // (e.msg) -> e(e1.e5) < property = ( boucle  ) >,
   // (e.msg) -> e(e1.e3_1);
   //
   //e(e1.e2_1) |-
   // (e.avalerCarte) -> e(e1.e4);
   //
   //e(e1.e5) |-
   // (start_Montant.call_Montant) -> e(e1_2.e6) < property = ( boucle  ) >;
   //
   //e(e1.e3_1) |-
   // (rcv_Retrait_recupCarte.emit_Retrait_restituerCarte) -> e(e2.e4);
   //
   //e(e1.e4) |- ;
   //
   //e(e1_2.e6) |-
   // (saisie_sel.e) -> e(e1_3.e6) < property = ( boucle  ) >;
   //
   //e(e2.e4) |-
   // (rcv_Retrait_resRetrait.emit_Retrait_resRetrait) -> e(e3.f);
```

```
//
//e(e1_3.e6) |-
// (emit_Montant_resMontant.rcv_Montant_resMontant) -> e(e1.e7) < property = ( boucle  ) >;
//
//e(e3.f) |- ;
//
//e(e1.e7) |-
// (e.msg) -> e(e1.e3) < property = ( boucle  ) >,
// (e.debiter) -> e(e1.e7_1);
//
//e(e1.e7_1) |-
// (rcv_Retrait_recupCarte.emit_Retrait_restituerCarte) -> e(e2.e8);
//
//e(e2.e8) |-
// (rcv_Retrait_resRetrait.emit_Retrait_resRetrait) -> e(e3.f);
//<
//initial = { e(init.init) },
//finaux = { e(e3.f) },
//dead = { e(e1.e4) },
//deadlock = { e(e1.e2_1), e(e1.e4) }
//>.
//stoplog();
//
```

B Tests

This appendix shows the detection of some protocol error by MEC.

B.1 Non déterminism and Inconsistencies in the Communications

B.1.1 Specification of the Synchronous Version

```
\*
      Author: P. ANDRE
      Project: COLOSS
      Experimentation : model-checking of services with MEC
      Case study : non déterminisme des échanges

      Version 1 : Systeme sequentiel sans gardes et parametres

      Creation date: 11/04/05
      Modification date: 11/04/05

      load(testChoix);
   *\

   transition_system ProcA < width = 0 >;
   e0 |- e -> e0,
        rcv_c_a -> e1,
        rcv_c_b -> e2;
   e1 |- e -> e1,
        i -> f; \* f est suppose final *\
   e2 |- e -> e2,
        i -> f; \* f est suppose final *\
   f |- e -> f;
   < initial = { e0 } ; final = {f} >.

   transition_system ProcB < width = 0 >;
   e0 |- e -> e0,
        emit_c_a -> e1,
        emit_c_b -> e2;
   e1 |- e -> e1,
        i -> f; \* f est suppose final *\
```

```
   e2 |- e -> e2,
        i -> f; \* f est suppose final *\
   f |- e -> f;
   < initial = { e0 } ; final = {f} >.

   synchronization_system testChoix < width = 2 ;
       list = ( ProcA, ProcB ) >;

   \* actions interne de l'automate ProcA *\
   ( i . e );
   \* actions interne de l'automate ProcB *\
   ( e . i );
   \* communications sur c *\
   ( rcv_c_a . emit_c_a );
   ( rcv_c_b . emit_c_b ).

   function inevitable(Y:trans ; X:state) return Z:state;
   begin
   Z = X \/ (src(Y /\ rtgt(Z)) - src(Y /\ rtgt(*-Z)))
   end.

   sync(testChoix, testChoix);
   dts(testChoix);
   finaux := final[1] /\ final[2];
   dead := (* - src(*)) - finaux;
   deadlock:=inevitable(*,dead);
   boucle := loop(*,*);
   log(testChoixLog);
   version();
   dts(testChoix);
   wts(*,*);
   stoplog();
```

B.1.2 Results of the Synchronous Version

```
version();
   //MEC 4 version alpha 3.4dts(testChoix);
   //Current St : testChoix
   //-- States :
   // * : 8
   //initial : 1
   //finaux : 1
   //dead : 0
   //deadlock : 0
   //-- Transitions :
   // * : 10
   //boucle : 0
   //wts(*,*);
   //transition_system testChoix
   //< width = 2;  list = (ProcA, ProcB )>;
   //
   //
   //e(e0.e0) |-
   // (rcv_c_a.emit_c_a) -> e(e1.e1),
   // (rcv_c_b.emit_c_b) -> e(e2.e2);
   //
   //e(e1.e1) |-
   // (e.i) -> e(e1.f),
   // (i.e) -> e(f.e1);
   //
   //e(e2.e2) |-
   // (e.i) -> e(e2.f),
   // (i.e) -> e(f.e2);
   //
   //e(e1.f) |-
```

```
// (i.e) -> e(f.f);
//
//e(f.e1) |-
// (e.i) -> e(f.f);
//
//e(e2.f) |-
// (i.e) -> e(f.f);
//
//e(f.e2) |-
// (e.i) -> e(f.f);
//
//e(f.f) |- ;
//<
//initial = { e(e0.e0) },
//finaux = { e(f.f) },
//dead = {  },
//deadlock = {  }
//>.
//stoplog();
//
```

B.1.3 Specification of the Asynchronous Version

```
\*
      Author: P. ANDRE
      Project: COLOSS
      Experimentation : model-checking of services with MEC
      Case study : non déterminisme des échanges dissocié

      Version 1 : Systeme sequentiel sans gardes et parametres

      Creation date: 11/04/05
      Modification date: 11/04/05

      load(testChoixND);
   *\

   transition_system ProcA < width = 0 >;
   e0 |- e -> e0,
        rcv_c -> e0_1,
        rcv_c -> e0_2;
   e0_1 |- e -> e0_1,
        a -> e1;
   e0_2 |- e -> e0_2,
        b -> e2;
   e1 |- e -> e1,
        i -> f; \* f est suppose final *\
   e2 |- e -> e2,
        i -> f; \* f est suppose final *\
   f |- e -> f;
   < initial = { e0 } ; final = {f} >.

   transition_system ProcB < width = 0 >;
   e0 |- e -> e0,
        emit_c -> e0_1,
        emit_c -> e0_2;
   e0_1 |- e -> e0_1,
        a -> e1;
   e0_2 |- e -> e0_2,
        b -> e2;
   e1 |- e -> e1,
        i -> f; \* f est suppose final *\
   e2 |- e -> e2,
        i -> f; \* f est suppose final *\
   f |- e -> f;
```

```
   < initial = { e0 } ; final = {f} >.

   synchronization_system testChoix < width = 2 ;
       list = ( ProcA, ProcB ) >;

   \* actions interne de l'automate ProcA *\
   ( i . e );
   \* actions interne de l'automate ProcB *\
   ( e . i );
   \* communications sur c *\
   ( rcv_c . emit_c );
   ( rcv_c . emit_c );
   \* échanges sur c *\
   ( a . a );
   ( b . b ).

   function inevitable(Y:trans ; X:state) return Z:state;
   begin
   Z = X \/ (src(Y /\ rtgt(Z)) - src(Y /\ rtgt(*-Z)))
   end.

   sync(testChoix, testChoix);
   dts(testChoix);
   finaux := final[1] /\ final[2];
   dead := (* - src(*)) - finaux;
   deadlock:=inevitable(*,dead);
   boucle := loop(*,*);
   log(testChoixNDLog);
   version();
   dts(testChoix);
   wts(*,*);
   stoplog();
```

B.1.4 Results of the Asynchronous Version

```
version();
   //MEC 4 version alpha 3.4dts(testChoix);
   //Current St : testChoix
   //-- States :
   // * : 12
   //initial : 1
   //finaux : 1
   //dead : 2
   //deadlock : 2
   //-- Transitions :
   // * : 14
   //boucle : 0
   //wts(*,*);
   //transition_system testChoix
   //< width = 2;  list = (ProcA, ProcB )>;
   //
   //
   //e(e0.e0) |-
   // (rcv_c.emit_c) -> e(e0_1.e0_1),
   // (rcv_c.emit_c) -> e(e0_1.e0_2),
   // (rcv_c.emit_c) -> e(e0_2.e0_1),
   // (rcv_c.emit_c) -> e(e0_2.e0_2);
   //
   //e(e0_1.e0_1) |-
   // (a.a) -> e(e1.e1);
   //
   //e(e0_1.e0_2) |- ;
   //
   //e(e0_2.e0_1) |- ;
   //
```

```
//e(e0_2.e0_2) |-
// (b.b) -> e(e2.e2);
//
//e(e1.e1) |-
// (e.i) -> e(e1.f),
// (i.e) -> e(f.e1);
//
//e(e2.e2) |-
// (e.i) -> e(e2.f),
// (i.e) -> e(f.e2);
//
//e(e1.f) |-
// (i.e) -> e(f.f);
//
//e(f.e1) |-
// (e.i) -> e(f.f);
//
//e(e2.f) |-
// (i.e) -> e(f.f);
//
//e(f.e2) |-
// (e.i) -> e(f.f);
//
//e(f.f) |- ;
//<
//initial = { e(e0.e0) },
//finaux = { e(f.f) },
//dead = { e(e0_1.e0_2), e(e0_2.e0_1) },
//deadlock = { e(e0_1.e0_2), e(e0_2.e0_1) }
//>.
//stoplog();
//
```

LABORATOIRE D'INFORMATIQUE DE NANTES-ATLANTIQUE

A Service-Based Component Model: Formalism, Analysis and Mechanization

P. André, G. Ardourel, C. Attiogbé, H. Habrias, C. Stoquer

Abstract

Component-Based Software Engineering (CBSE) is one of the approaches to master the development of large scale software. In this setting, the verification concern is still a challenge. The objective of our work is to provide the designer of components-based systems with the methods to assist his/her use of the components. In particular, the current work adresses the composability of components and their services.
A component model is presented, based on services. An associated simple but expressive formalism is introduced; it describes the services as extended LTS and their structuring as components. The composition of components is mainly based on service composition and encapsulation.
The composability of component is defined from the composability of services. To ensure the correctness of component composition, we check that an assembly is possible via the checking of the composabiblity of the linked services, and their behavioral compatibility. In order to mechanize our approach, the services and the components are translated into the MEC and LOTOS formalism. Finally the MEC and LOTOS CADP toolbox is used to perform experiments.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; D.2.11 [**Software Engineering**]: Software Architectures—*Languages*

General Terms: Components, Services, Behavioural Interface Description, Interaction Checking

Additional Key Words and Phrases: Components, Services, Behavioural Interface Description, Interaction Checking

LINA, Université de Nantes
2, rue de la Houssinière
B.P. 92208 — F-44322 NANTES CEDEX 3